

Exhibit A

```
// CoLCBroker.h : Declaration of the CoLCBroker
```

```
#ifndef LCBROKER_H
#define __LCBROKER_H__
```

```
#include "resource.h" // main symbols
#include "errorMessage.h"
#include "xcLCBroker.h"
#include "ISLXmlCmdsImpl.h"
```

```
////////////////////////////////////
```

```
_COM_SMARTPTR_TYPEDEF(ISLXmlCmds, __uuidof(ISLXmlCmds));
```

```
////////////////////////////////////
```

```
// CoLCBroker
```

```
class CSdoConnection;
class CXmlDocument;
class CIdServer;
class CIdGenAudit;
class CIdGenCpi;
class CIdGenEncounter;
class CIdGenUnregUser;
```

```
class ATL_NO_VTABLE CoLCBroker :
```

```
public CComObjectRootEx<CComMultiThreadModel>,
public CComCoClass<CoLCBroker, &CLSID_LCBroker>,
public ISupportErrorInfoImpl<&IID_ISLXmlCmds>,
// public ISLXmlControl2Impl<CxcLCBrokerFactory>
public IDispatchImpl<ISLXmlCmdsImpl<CxcLCBrokerFactory>, &IID_ISLXmlCmds, &
LIBID_LCBROKERLib>
```

```
{
public:
```

```
CoLCBroker()
{
}
```

```
HRESULT FinalConstruct();
void FinalRelease();
```

```
DECLARE_REGISTRY_RESOURCEID(IDR_LCBroker)
DECLARE_NOT_AGGREGATABLE(CoLCBroker)
```

```
DECLARE_PROTECT_FINAL_CONSTRUCT()
```

```
BEGIN_COM_MAP(CoLCBroker)
```

```
COM_INTERFACE_ENTRY(ISLXmlCmds)
COM_INTERFACE_ENTRY(ISupportErrorInfo)
COM_INTERFACE_ENTRY(IDispatch)
```

```
END_COM_MAP()
```

```
// data members
```

```
protected:
```

```
ISLXmlControl2Ptr m_spSearcher;
CSdoConnection * m_pconnSdo;
CIdServer * m_pserverId;
CIdGenAudit * m_pidgenAudit;
CIdGenCpi * m_pidgenCpi;
CIdGenEncounter * m_pidgenEncounter;
CIdGenUnregUser * m_pidgenUnregUser;
CErrorMessage m_emLast;
```

```
string m_strAlias;
string m_strHostName;
```

```
// ISLXmlCmds override
```

```
virtual bool execXmlCmd(CXmlDocument & docXmlCmd, CXmlDocument ** ppdocXmlResult, string & strError);
```

```
// internal C++ interface
```

```
public:
```

```
    bool        setConnection(CSdoConnection * pconnSdo);
    void        destroyConnection();
    ISLXmlControl2* getSearcher();
    CSdoConnection* getConnection();
    CIdGenAudit*  getAuditIdGenerator(){return m_pidgenAudit;}
    CIdGenCpi*    getCpiIdGenerator(){return m_pidgenCpi;}
    CIdGenEncounter* getEncIdGenerator(){return m_pidgenEncounter;}
    CIdGenUnregUser* getUnregUserIdGenerator(){return m_pidgenUnregUser;}
    void        getLastError(string & strError){m_emLast.getError(strError);}
    CErrorMessage& getLastError(){return m_emLast;}
    void        setAlias(string strAlias) { m_strAlias = strAlias; }
    void        getDefaultAlias();
    const char*  getHostName() { return m_strHostName.c_str(); }
```

```
};
```

```
#endif // __LCBROKER_H_
```

```
/*//////////////////////////////////////
//////////////////////////////////////
Encrypt/ Decrypt Rotines
//////////////////////////////////////

Dependencies :

#include <string>
#include <list>
#include <fstream>
#include <strstream>
using namespace std;

//////////////////////////////////////
//////////////////////////////////////*/

#ifndef _ENCRYPTOR_H
#define _ENCRYPTOR_H

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

class CEncryptor
{
protected:
    string m_strDefaultKey;

    short AsciiHexToInt( LPCTSTR pszString, int* pnAnswer );
    short AsciiHexToInt( string& strIn, int* pnAnswer )
        {return AsciiHexToInt(strIn.c_str(), pnAnswer);}

public:
    CEncryptor();
    bool Encrypt(LPCTSTR pszIn, LPCSTR psKey, string & strOut);
    bool Decrypt(LPCTSTR pszIn, LPCSTR psKey, string & strOut);
};

#endif // _ENCRYPTOR_H
```



```
#ifndef errorMessage_h
#define _errorMessage_h

class CErrorMessage : public std::stringstream
{
public:
    void appendError(_com_error & e)
    {
        string strError = (char *) e.Description();
        HRESULT hr = e.Error();
        *this << "COM Error = [" << strError << "]. hr = [" << std::hex << hr << "].";
        return;
    }

    void appendError(HRESULT hr)
    {
        *this << "hr = [" << std::hex << hr << std::dec << "].";
        return;
    }

    void appendError(CErrorMessage & em)
    {
        appendError((std::stringstream &) em);
    }

    void appendError(std::stringstream & strmError)
    {
        strmError << '\0';
        *this << strmError.str();
        strmError.freeze(false);
    }

    void setError(_com_error & e)
    {
        clear();
        appendError(e);
    }

    void setError(HRESULT hr)
    {
        clear();
        appendError(hr);
    }

    void setError(LPCSTR pszError)
    {
        clear();
        *this << pszError;
    }

    void setError(CErrorMessage & em)
    {
        clear();
        appendError(em);
    }

    void getError(string & strError)
    {
        *this << '\0';
        strError = str();
        freeze(false);
        return;
    }

    string getError()
    {
        string strError;
    }
}
```

```
        *this << '\0';
        strError = str();
        freeze(false);
        return strError;
    }

    void getError(std::stringstream & strmError)
    {
        *this << '\0';
        strmError << str();
        freeze(false);
        return;
    }

    void clear()
    {
        seekp(0);
    }
};

#endif
```

```
#ifndef idGen_h
#define _idGen_h

#include "idGenBase.h"
#include "rs_audit.h"

class CIdGenAudit : public CIdGenerator
{
protected:
    Crs_audit    m_rsAudit;

public:
    CIdGenAudit(CIdServer * pidServer);
    virtual long generateId(DWORD dwMsgNo, DWORD dwUserId = 0, LPCSTR pszHost = NULL,
        LPCSTR pszAppl = NULL);
};

class CIdGenCpi : public CIdGenerator
{
public:
    CIdGenCpi(CIdServer *pidServer);
    virtual long generateId();
};

class CIdGenEncounter : public CIdGenerator
{
public:
    CIdGenEncounter(CIdServer *pidServer);
    virtual long generateId();
};

class CIdGenUnregUser: public CIdGenerator
{
public:
    CIdGenUnregUser(CIdServer *pidServer);
    virtual long generateId();
};

#endif
```

```

#ifndef _ISLXmlCmds_h
#define _ISLXmlCmds_h

#include "xmlParser.h"
#include "xmlCommand.h"

#ifdef _DEBUG
#define _DUMP_XML
#endif

/*
    This file is a templated implementation of the ISLXmlCmds interface. The template
    argument should be a class derived from CXmlCommandFactory. The class will call the command
    factory to instantiate command processors requested by callers, and call the execute() method on
    those processors.
*/

/*
XCF = xml command factory
*/

template<class XCF>
class ISLXmlCmdsImpl : public ISLXmlCmds
{
public:
    //////////////////////////////////////
    // ISLXmlCmds interface
    STDMETHODIMP Exec(VARIANT vXMLCmd, VARIANT * pvError, VARIANT * pvXMLResults);
    STDMETHODIMP ExecSet(VARIANT vXMLCmd, VARIANT * pvError, VARIANT * pvXMLResults);
    STDMETHODIMP ExecTest();

protected:
    //////////////////////////////////////
    // internal C++ interface (overridable)
    virtual bool execXmlCmdSet(BSTR bstrXMLCmd, BSTR * pbstrResults, BSTR * pbstrError);
    virtual bool execXmlCmd(BSTR bstrXMLCmd, BSTR * pbstrResults, BSTR * pbstrError);
    virtual bool execXmlCmd(CXmlDocument & docXmlCmd, CXmlDocument ** ppdocXmlResult,
        string & strError);

protected:
    // these variables should be set by the implementing coclass
    bool m_fConnected; // true if coclass is connected to
    database
    bool m_fCheckConnRequired; // true performs a check for dbconnection
    requirement.
    CComObjectRoot * m_pcoOwner; // the this pointer of the coclass
    CLogBase * m_plogXml; // a place to dump debug xml
};

////////////////////////////////////
// implementation
////////////////////////////////////

////////////////////////////////////
// ISLXmlControl interface
////////////////////////////////////

/*****
FUNCTION: ExecText

CLASS: ISLXmlCmdsImpl<XCF>

DESCRIPTION: Takes an XML command, executes it, and returns the results of the
command in a BSTR.
*****/

```

PARAMETERS: bstrXMLCmd - the XML command to execute

pbstrError - a pointer to BSTR that receives a verbose error message

pbstrXMLResults - a pointer to a BSTR that receives the results
of the XML command in XML.

RETURNS: S_OK, if an error occurs the out param pbstrError will be valued.

```

*****/
template<class XCF>
STDMETHODIMP ISLXmlCmdsImpl<XCF>::Exec(VARIANT vXMLCmd, VARIANT * pvError, VARIANT *
pvXMLResults)

```

```

{
    BSTR bstrError = NULL;
    BSTR bstrResults = NULL;

    VariantInit(pvError);
    VariantInit(pvXMLResults);

    _bstr_t bstrXMLCmd(vXMLCmd);

    bool fSuccess = execXmlCmd(bstrXMLCmd, &bstrResults, &bstrError);

    V_VT(pvError) = VT_BSTR;
    V_BSTR(pvError) = bstrError;
    V_VT(pvXMLResults) = VT_BSTR;
    V_BSTR(pvXMLResults) = bstrResults;

    return S_OK;
}

```

```

/*****
    FUNCTION: ExecSet

```

```

    CLASS: ISLXmlCmdsImpl<XCF>

```

DESCRIPTION: Takes a set of XML commands and executes each one in succession.
The results from each command is adopted by the result's root. The
resulting xml text is returned in a BSTR.

PARAMETERS: bstrXMLCmd - The command set to execute.

```

ex: <commandset>
    <command name="someCommand">
        <parm name="someParm">99</parm>
    </command>
    ...
</commandset>

```

pbstrError - a pointer to BSTR that receives a verbose error message

pbstrXMLResults - a pointer to a BSTR that receives the results
of the XML command in XML.

RETURNS: S_OK, if an error occurs the out param pbstrError will be valued.

```

*****/
template<class XCF>
STDMETHODIMP ISLXmlCmdsImpl<XCF>::ExecSet(VARIANT vXMLCmd, VARIANT * pvError, VARIANT *
pvXMLResults)

```

```

{
    BSTR bstrError = NULL;
    BSTR bstrResults = NULL;

```

```

VariantInit(pvError);
VariantInit(pvXMLResults);

_bstr_t bstrXMLCmd(vXMLCmd);

bool fSuccess = execXmlCmdSet(bstrXMLCmd, &bstrResults, &bstrError);

V_VT(pvError) = VT_BSTR;
V_BSTR(pvError) = bstrError;
V_VT(pvXMLResults) = VT_BSTR;
V_BSTR(pvXMLResults) = bstrResults;

return S_OK;
}

/*****
FUNCTION: ExecTest

CLASS: ISLXmlCmdsImpl<XCF>

DESCRIPTION: Can be used for quick and dirty testing.

*****/
template<class XCF>
STDMETHODIMP ISLXmlCmdsImpl<XCF>::ExecTest()
{
    CLogMsg msgTest("Hey from ExecTest");
    msgTest.Post(_logFile);

    HRESULT hr = S_OK;
    return hr;
}

////////////////////////////////////
// internal C++ interface
////////////////////////////////////
/*****
FUNCTION: execXmlCmdSet

CLASS: ISLXmlCmdsImpl<XCF>

DESCRIPTION: Executes each command contained in an xml command set. The xml is
in the following format.

    <commandset>
        <command name="someName">
            <parm name="someName">parmValue</parm>
        </command>
        ...
    </commandset>

PARAMETERS: bstrXMLCmd - BSTR containing the xml command set.

pbstrResults - pointer to a BSTR that will receive the results of
the command. If NULL, then results are expected to
go into the following parameter.

pbstrError - pointer to a BSTR that will receive a verbose message
if some error occurs.

RETURNS: true on success

*****/
template<class XCF>
bool ISLXmlCmdsImpl<XCF>::execXmlCmdSet(BSTR bstrXMLCmd, BSTR * pbstrResults, BSTR *
pbstrError)

```

```
{
    bool fSuccess = true;
    string strError;

    string strXmlCmds = (char *) _bstr_t(bstrXMLCmd);

#ifdef _DUMP_XML
    if (m_plogXml)
    {
        CLogMsg msg;

        msg << "++++++++++++++++++++";
        msg.Post(*m_plogXml);
        msg.Clear();
        msg << "Command = [" << strXmlCmds.c_str() << "];";
        msg.Post(*m_plogXml);
    }
#endif

    // parse the xml
    CXmlDocument docXml(strXmlCmds.c_str());
    if (!docXml.isReady())
    {
        string strParseError;
        docXml.getParserError(strParseError);
        strError = "Parse of XML failed. Error = [";
        strError += strParseError;
        strError += "];";
        fSuccess = false;
    }

    // make sure it is a valid command set
    CXmlElement elRoot;
    docXml.getRoot(&elRoot);
    string strTag;
    elRoot.getTag(strTag);
    if (strcmp(strTag.c_str(), "commandset") != 0)
    {
        strError = "Document tag must be \"commandset\".";
        fSuccess = false;
    }

    // each child of the root should be a command, execute each one
    CXmlDocument * pdocResults = NULL;
    if (fSuccess)
    {
        pdocResults = new CXmlDocument("<commandset/>");

        CXmlElement elCmd;
        bool fCmd = elRoot.getFirst(&elCmd);
        while (fCmd && fSuccess)
        {
            docXml.pushCurrent(&elCmd);
            fSuccess = execXmlCmd(docXml, &pdocResults, strError);
            docXml.popCurrent();
            fCmd = elRoot.getNext(&elCmd);
        }
    }

    // not all commands return results
    if (fSuccess && pdocResults != NULL)
    {
        string strResults;
        pdocResults->getXML(strResults);
        *pbstrResults = _bstr_t(strResults.c_str()).copy();

#ifdef _DUMP_XML
```

```

        if (m_plogXml)
        {
            string strDump;
            pdocResults->getXML(strDump);
            CLogMsg msg;
            msg << "\n\nResults = [\n" << strDump.c_str() << "\n]";
            msg.Post(*m_plogXml);
        }
    #endif
}

// report any error information
if (strError.size())
{
    *pbstrError = _bstr_t(strError.c_str()).copy();

    #ifdef _DUMP_XML
        if (m_plogXml)
        {
            CLogMsg msg;
            msg << "\n\nError = [" << strError << "\n]";
            msg.Post(*m_plogXml);
        }
    #endif
}

// clean up
if (pdocResults != NULL)
    delete pdocResults;

return fSuccess;
}

/*****
FUNCTION: execXmlCmd

CLASS: ISLXmlCmdsImpl<XCF>

DESCRIPTION: Executes one XML command. The XML is in the following format.

        <command name="someName">
            <parm name="someName">parm value</parm>
        </command>

PARAMETERS: bstrXMLCmd - BSTR containing the xml command.

        pbstrResults - pointer to a BSTR that will receive the results of
                        the command. If NULL, then results are expected to
                        go into the following parameter.

        pbstrError - pointer to a BSTR that will receive a verbose message
                     if some error occurs.

RETURNS: true on success

*****/
template<class XCF>
bool ISLXmlCmdsImpl<XCF>::execXmlCmd(BSTR bstrXMLCmd, BSTR * pbstrResults, BSTR *
pbstrError)
{
    bool fSuccess = true;
    string strError;

    string strXmlCmd = (char *) _bstr_t(bstrXMLCmd);

    #ifdef _DUMP_XML

```



```

    if (m_plogXml)
    {
        CLogMsg msg;

        msg << "++++++++++++++++++++";
        msg.Post(*m_plogXml);
        msg.Clear();
        msg << "Command = [" << strXmlCmd.c_str() << "];";
        msg.Post(*m_plogXml);
    }
#endif

// parse the xml
CXmlDocument docXml(strXmlCmd.c_str());
if (!docXml.isReady())
{
    string strParseError;
    docXml.getParserError(strParseError);
    strError = "Parse of XML failed. Error = [";
    strError += strParseError;
    strError += "];";
    fSuccess = false;
}

// execute the command
CXmlDocument * pdocResults = NULL;
if (fSuccess)
    fSuccess = execXmlCmd(docXml, &pdocResults, strError);

// not all commands return results
if (fSuccess && pdocResults != NULL)
{
    string strResults;
    pdocResults->getXML(strResults);
    *pbstrResults = _bstr_t(strResults.c_str()).copy();

#ifdef _DUMP_XML
    if (m_plogXml)
    {
        string strDump;
        pdocResults->getXML(strDump);
        CLogMsg msg;
        msg << "\n\nResults = [\n" << strDump.c_str() << "];";
        msg.Post(*m_plogXml);
    }
#endif
}

// report any error information
if (strError.size())
{
    *pbstrError = _bstr_t(strError.c_str()).copy();

#ifdef _DUMP_XML
    if (m_plogXml)
    {
        CLogMsg msg;
        msg << "\n\nError = [" << strError << "];";
        msg.Post(*m_plogXml);
    }
#endif
}

// clean up
if (pdocResults != NULL)
    delete pdocResults;

```

```

    return fSuccess;
}

/*****
FUNCTION: execXmlCmd

CLASS: ISLXmlCmdsImpl<XCF>

DESCRIPTION: description text here.

PARAMETERS: docXmlCmd - xml document containing the command at the current node

               ppdocXmlResult - a pointer to a pointer of the result xml. If this parm
                               points to a null, then a new xml doc is created for the
                               caller. If this parm points to an xml document, then the
                               results are added as a child node to that document.

               strError - receives verbose error information.

RETURNS: true on success
*****/
template<class XCF>
bool ISLXmlCmdsImpl<XCF>::execXmlCmd(CXmlDocument & docXmlCmd, CXmlDocument **
    ppdocXmlResult,
                                   string & strError)
{
    bool fSuccess = true;

    // get xml command processor
    XCF xcFactory;
    CXmlCommand * pcmdXml = xcFactory.createCommand(&docXmlCmd);
    if (pcmdXml == NULL)
    {
        xcFactory.getLastErrorMessage(strError);
        fSuccess = false;
    }

    // make sure we have db connection if command requires it
    if (fSuccess && m_fCheckConnRequired)
    {
        if (pcmdXml->isConnectionRequired() && !m_fConnected)
        {
            strError = "There is no ADO connection. \"openDatabase\" must be executed
first.";
            fSuccess = false;
        }
    }

    // execute the command
    if (fSuccess)
    {
        // give command access to the coclass
        pcmdXml->setOwner(m_pcoOwner);

        if (fSuccess = pcmdXml->execCommand())
        {
            if (*ppdocXmlResult != NULL)
            {
                CXmlElement elRoot;
                CXmlDocument * pdoc = pcmdXml->getResults();
                if (pdoc != NULL)
                {
                    pdoc->getRoot(&elRoot);
                    (*ppdocXmlResult)->addChild(&elRoot);
                }
            }
        }
    }
}

```

```
        else
            *ppdocXmlResult = pcmdXml->getResults(true);
    }
    else
        pcmdXml->getLastError(strError);
}

// clean up
delete pcmdXml;

return fSuccess;
}

#endif
```

```
// LCBroker.cpp : Implementation of WinMain

// Note: Proxy/Stub Information
//      To build a separate proxy/stub DLL,
//      run nmake -f LCBrokerps.mk in the project directory.

#include "stdafx.h"
#include "resource.h"
#include <initguid.h>
#include "LCBroker.h"

#include "LCBroker_i.c"

#include <stdio.h>
#include "CoLCBroker.h"
#include "xmlCommand.h"
#include "xmlParser.h"
#include "registryDB.h"

CServiceModule _Module;

//////////////////////////////////////
//Global decalrations

CLogNTEvents    _logEvents("Lifeclinic Broker");
CLogFile        _logFile("c:\\LCBroker.log");
CLogDebug       _logDebug;
CLogMulti       _logAll;
string          _strDefaultAlias;
//////////////////////////////////////

BEGIN_OBJECT_MAP(ObjectMap)
    OBJECT_ENTRY(CLSID_LCBroker, CoLCBroker)
END_OBJECT_MAP()

LPCTSTR FindOneOf(LPCTSTR p1, LPCTSTR p2)
{
    while (p1 != NULL && *p1 != NULL)
    {
        LPCTSTR p = p2;
        while (p != NULL && *p != NULL)
        {
            if (*p1 == *p)
                return CharNext(p1);
            p = CharNext(p);
        }
        p1 = CharNext(p1);
    }
    return NULL;
}

// Although some of these functions are big they are declared inline since they are only ✓
// used once

inline HRESULT CServiceModule::RegisterServer(BOOL bRegTypeLib, BOOL bService)
{
    HRESULT hr = CoInitialize(NULL);
    if (FAILED(hr))
        return hr;

    // Remove any previous service since it may point to
    // the incorrect file
    Uninstall();

    // Add service entries
    UpdateRegistryFromResource(IDR_LCBroker, TRUE);
}
```

```
// Adjust the AppID for Local Server or Service
CRegKey keyAppID;
LONG lRes = keyAppID.Open(HKEY_CLASSES_ROOT, _T("AppID"), KEY_WRITE);
if (lRes != ERROR_SUCCESS)
    return lRes;

CRegKey key;
lRes = key.Open(keyAppID, _T("{D9DCC3F4-DE3C-11d3-B87B-8E0DB3000000}"), KEY_WRITE);
if (lRes != ERROR_SUCCESS)
    return lRes;
key.DeleteValue(_T("LocalService"));

if (bService)
{
    key.SetValue(_T("SLMD LCBroker"), _T("LocalService"));
    key.SetValue(_T("-Service"), _T("ServiceParameters"));
    // Create service
    Install();
}

// Add object entries
hr = CComModule::RegisterServer(bRegTypeLib);

CoUninitialize();
return hr;
}

inline HRESULT CServiceModule::UnregisterServer()
{
    HRESULT hr = CoInitialize(NULL);
    if (FAILED(hr))
        return hr;

    // Remove service entries
    UpdateRegistryFromResource(IDR_LCBroker, FALSE);
    // Remove service
    Uninstall();
    // Remove object entries
    CComModule::UnregisterServer(TRUE);
    CoUninitialize();
    return S_OK;
}

inline void CServiceModule::Init(_ATL_OBJMAP_ENTRY* p, HINSTANCE h, UINT nServiceNameID,
    const GUID* plibid)
{
    CComModule::Init(p, h, plibid);

    m_bService = TRUE;

    LoadString(h, nServiceNameID, m_szServiceName, sizeof(m_szServiceName) / sizeof
        (TCHAR));

    // set up the initial service status
    m_hServiceStatus = NULL;
    m_status.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
    m_status.dwCurrentState = SERVICE_STOPPED;
    m_status.dwControlsAccepted = SERVICE_ACCEPT_STOP;
    m_status.dwWin32ExitCode = 0;
    m_status.dwServiceSpecificExitCode = 0;
    m_status.dwCheckPoint = 0;
    m_status.dwWaitHint = 0;

    GetLocalTime(&m_statsLCBroker.m_systimeStarted);
    SystemTimeToVariantTime(&m_statsLCBroker.m_systimeStarted, &m_statsLCBroker.
        m_vartimeStarted);
```

```
}

LONG CServiceModule::Unlock()
{
    LONG l = CComModule::Unlock();
    if (l == 0 && !m_bService)
        PostThreadMessage(dwThreadId, WM_QUIT, 0, 0);
    return l;
}

BOOL CServiceModule::IsInstalled()
{
    BOOL bResult = FALSE;

    SC_HANDLE hSCM = ::OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);

    if (hSCM != NULL)
    {
        SC_HANDLE hService = ::OpenService(hSCM, m_szServiceName, SERVICE_QUERY_CONFIG);
        if (hService != NULL)
        {
            bResult = TRUE;
            ::CloseServiceHandle(hService);
        }
        ::CloseServiceHandle(hSCM);
    }
    return bResult;
}

inline BOOL CServiceModule::Install()
{
    if (IsInstalled())
        return TRUE;

    SC_HANDLE hSCM = ::OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
    if (hSCM == NULL)
    {
        MessageBox(NULL, _T("Couldn't open service manager"), m_szServiceName, MB_OK);
        return FALSE;
    }

    // Get the executable file path
    TCHAR szFilePath[_MAX_PATH];
    ::GetModuleFileName(NULL, szFilePath, _MAX_PATH);

    SC_HANDLE hService = ::CreateService(
        hSCM, m_szServiceName, m_szServiceName,
        SERVICE_ALL_ACCESS, SERVICE_WIN32_OWN_PROCESS,
        SERVICE_DEMAND_START, SERVICE_ERROR_NORMAL,
        szFilePath, NULL, NULL, _T("RPCSS\0"), NULL, NULL);

    if (hService == NULL)
    {
        ::CloseServiceHandle(hSCM);
        MessageBox(NULL, _T("Couldn't create service"), m_szServiceName, MB_OK);
        return FALSE;
    }

    ::CloseServiceHandle(hService);
    ::CloseServiceHandle(hSCM);
    return TRUE;
}

inline BOOL CServiceModule::Uninstall()
{
    if (!IsInstalled())
        return TRUE;
```

```

SC_HANDLE hSCM = ::OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);

if (hSCM == NULL)
{
    MessageBox(NULL, _T("Couldn't open service manager"), m_szServiceName, MB_OK);
    return FALSE;
}

SC_HANDLE hService = ::OpenService(hSCM, m_szServiceName, SERVICE_STOP | DELETE);

if (hService == NULL)
{
    ::CloseServiceHandle(hSCM);
    MessageBox(NULL, _T("Couldn't open service"), m_szServiceName, MB_OK);
    return FALSE;
}

SERVICE_STATUS status;
::ControlService(hService, SERVICE_CONTROL_STOP, &status);

BOOL bDelete = ::DeleteService(hService);
::CloseServiceHandle(hService);
::CloseServiceHandle(hSCM);

if (bDelete)
    return TRUE;

MessageBox(NULL, _T("Service could not be deleted"), m_szServiceName, MB_OK);
return FALSE;
}

```

```

////////////////////////////////////
// Logging functions
void CServiceModule::LogEvent(LPCTSTR pFormat, ...)
{

```

```

    TCHAR chMsg[2048];
    va_list pArg;

    va_start(pArg, pFormat);
    _vstprintf(chMsg, pFormat, pArg);
    va_end(pArg);

    CLogMsgEvent(HL7EV_GENERAL_FAILURE, -1, chMsg).Post(_logAll);
}

```

```

////////////////////////////////////✓
////

```

```

// Service startup and registration
inline void CServiceModule::Start()
{

```

```

    SERVICE_TABLE_ENTRY st[] =
    {
        { m_szServiceName, _ServiceMain },
        { NULL, NULL }
    };
    if (m_bService && !::StartServiceCtrlDispatcher(st))
    {
        m_bService = FALSE;
    }
    if (m_bService == FALSE)
        Run();
}

```

```

inline void CServiceModule::ServiceMain(DWORD /* dwArgc */, LPTSTR* /* lpszArgv */)
{
    // Register the control request handler
    m_status.dwCurrentState = SERVICE_START_PENDING;

```

```
    m_hServiceStatus = RegisterServiceCtrlHandler(m_szServiceName, _Handler);
    if (m_hServiceStatus == NULL)
    {
        LogEvent(_T("Handler not installed"));
        return;
    }
    SetServiceStatus(SERVICE_START_PENDING);

    m_status.dwWin32ExitCode = S_OK;
    m_status.dwCheckPoint = 0;
    m_status.dwWaitHint = 0;

    // When the Run function returns, the service has stopped.
    Run();

    SetServiceStatus(SERVICE_STOPPED);
    CLogMsgEvent(HL7EV_SERVICE_STOPPED).Post(_logAll);
}

inline void CServiceModule::Handler(DWORD dwOpcode)
{
    switch (dwOpcode)
    {
    case SERVICE_CONTROL_STOP:
        SetServiceStatus(SERVICE_STOP_PENDING);
        PostThreadMessage(dwThreadId, WM_QUIT, 0, 0);
        break;
    case SERVICE_CONTROL_PAUSE:
        break;
    case SERVICE_CONTROL_CONTINUE:
        break;
    case SERVICE_CONTROL_INTERROGATE:
        break;
    case SERVICE_CONTROL_SHUTDOWN:
        break;
    default:
        LogEvent(_T("Bad service request"));
    }
}

void WINAPI CServiceModule::_ServiceMain(DWORD dwArgc, LPTSTR* lpszArgv)
{
    _Module.ServiceMain(dwArgc, lpszArgv);
}

void WINAPI CServiceModule::_Handler(DWORD dwOpcode)
{
    _Module.Handler(dwOpcode);
}

void CServiceModule::SetServiceStatus(DWORD dwState)
{
    m_status.dwCurrentState = dwState;
    ::SetServiceStatus(m_hServiceStatus, &m_status);
}

void CServiceModule::Run()
{
    _Module.dwThreadId = GetCurrentThreadId();

    // HRESULT hr = CoInitialize(NULL);
    // If you are running on NT 4.0 or higher you can use the following call
    // instead to make the EXE free threaded.
    // This means that calls come in on a random RPC thread
    HRESULT hr = CoInitializeEx(NULL, COINIT_MULTITHREADED);

    _ASSERT(SUCCEEDED(hr));
}
```



```

// This provides a NULL DACL which will allow access to everyone.
CSecurityDescriptor sd;
sd.InitializeFromThreadToken();
hr = CoInitializeSecurity(sd, -1, NULL, NULL,
    RPC_C_AUTHN_LEVEL_PKT, RPC_C_IMP_LEVEL_IMPERSONATE, NULL, EOAC_NONE, NULL);
_ASSERTE(SUCCEEDED(hr));

hr = _Module.RegisterClassObjects(CLSCTX_LOCAL_SERVER | CLSCTX_REMOTE_SERVER,
    REGCLS_MULTIPLEUSE);
_ASSERTE(SUCCEEDED(hr));

CLogMsgEvent("Service Started").Post(_logAll);
if (m_bService)
    SetServiceStatus(SERVICE_RUNNING);

////////////////////////////////////////
//init default registry Alias.
////////////////////////////////////////
_strDefaultAlias = "";

MSG msg;
while (GetMessage(&msg, 0, 0, 0))
    DispatchMessage(&msg);

CLogMsgEvent("Service Stopped").Post(_logAll);

_Module.RevokeClassObjects();

CoUninitialize();
}

////////////////////////////////////////
//
extern "C" int WINAPI _tWinMain(HINSTANCE hInstance,
    HINSTANCE /*hPrevInstance*/, LPTSTR lpCmdLine, int /*nShowCmd*/)
{
    _logAll.AddLog(&_logEvents);
    _logDebug.Enabled(false);
    _logAll.AddLog(&_logFile);

#ifdef _DEBUG
    _logEvents.EnableTranslation(true);
    _logDebug.Enabled(true);
    _logAll.AddLog(&_logDebug);
#endif

    lpCmdLine = GetCommandLine(); //this line necessary for _ATL_MIN_CRT
    _Module.Init(ObjectMap, hInstance, IDS_SERVICENAME, &LIBID_LCBROKERLib);
    _Module.m_bService = TRUE;

    TCHAR szTokens[] = _T("-/");

    LPCTSTR lpszToken = FindOneOf(lpCmdLine, szTokens);
    while (lpszToken != NULL)
    {
        if (lstrcmpi(lpszToken, _T("UnregServer"))==0)
            return _Module.UnregisterServer();

        // Register as Local Server
        if (lstrcmpi(lpszToken, _T("RegServer"))==0)
            return _Module.RegisterServer(TRUE, FALSE);

        // Register as Service
        if (lstrcmpi(lpszToken, _T("Service"))==0)
            return _Module.RegisterServer(TRUE, TRUE);

        lpszToken = FindOneOf(lpszToken, szTokens);
    }
}

```

```
}

// Are we Service or Local Server
CRegKey keyAppID;
LONG lRes = keyAppID.Open(HKEY_CLASSES_ROOT, _T("AppID"), KEY_READ);
if (lRes != ERROR_SUCCESS)
    return lRes;

CRegKey key;
lRes = key.Open(keyAppID, _T("{75751D72-AFD1-11D2-AC59-00C04F6E4C48}"), KEY_READ);
if (lRes != ERROR_SUCCESS)
    return lRes;

TCHAR szValue[_MAX_PATH];
DWORD dwLen = _MAX_PATH;
lRes = key.QueryValue(szValue, _T("LocalService"), &dwLen);

_Module.m_bService = FALSE;
if (lRes == ERROR_SUCCESS)
    _Module.m_bService = TRUE;

_Module.Start();

// When we get here, the service has been stopped
return _Module.m_status.dwWin32ExitCode;
}
```

```
#include "stdafx.h"
#include "Logging.h"
#include "Registry.h"

////////////////////////////////////////
////////////////////////////////////////
// Log messages
////////////////////////////////////////
////////////////////////////////////////
CLogMsg::CLogMsg()
{
    m_pszText = NULL;
}

CLogMsg::CLogMsg(LPCSTR pszMessage)
{
    m_pszText = NULL;
    if (pszMessage != NULL)
        *this << pszMessage;
}

CLogMsg::CLogMsg(string & strMessage)
{
    m_pszText = NULL;
    *this << strMessage;
}

CLogMsg::~CLogMsg()
{
    ReleaseBuffers();
}

CLogMsg & CLogMsg::Format(LPCSTR pszFormat, ...)
{
    Clear();
    va_list pArgs;
    va_start(pArgs, pszFormat);
    TCHAR pszBuffer [1024];
    vsprintf(pszBuffer, pszFormat, pArgs);
    va_end(pArgs);
    *this << pszBuffer;
    return *this;
}

void CLogMsg::Post(CLogBase & log)
{
    log.Post(this);
    return;
}

long CLogMsg::Event()
{
    return 0;
}

long CLogMsg::Severity()
{
    return EVENTLOG_SUCCESS;
}

TCHAR ** CLogMsg::Arguments(long * plArgCount)
{
    *plArgCount = 1;
    Text();
    return &m_pszText;
}
```

```
TCHAR * CLogMsg::Text()
{
    ReleaseBuffers();
    *this << '\0';
    TCHAR * pszText = str();
    int nLen = pcount();
    m_pszText = new TCHAR [nLen + 1];
    _tcscpy(m_pszText, pszText);
    freeze(false);
    return m_pszText;
}

void CLogMsg::ReleaseBuffers()
{
    if (m_pszText != NULL)
    {
        delete [] m_pszText;
        m_pszText = NULL;
    }
    return;
}

void CLogMsg::Clear()
{
    ReleaseBuffers();
    seekp(0);
    return;
}

void CLogMsg::appendError(_com_error & e)
{
    string strError = (char *) e.Description();
    HRESULT hr = e.Error();
    *this << "COM Error = [" << strError << "]. hr = [" << std::hex << hr << "].";
    return;
}

void CLogMsg::appendError(HRESULT hr)
{
    *this << "hr = [" << std::hex << hr << std::dec << "].";
    return;
}

void CLogMsg::appendError(CLogMsg & em)
{
    appendError((std::stringstream &) em);
}

void CLogMsg::appendError(std::stringstream & strmError)
{
    strmError << '\0';
    *this << strmError.str();
    strmError.freeze(false);
}

void CLogMsg::setError(_com_error & e)
{
    clear();
    appendError(e);
}

void CLogMsg::setError(HRESULT hr)
{
    clear();
    appendError(hr);
}
```

```
void CLogMsg::setError(LPCSTR pszError)
{
    clear();
    *this << pszError;
}

void CLogMsg::setError(CLogMsg & em)
{
    clear();
    appendError(em);
}

void CLogMsg::getError(string & strError)
{
    *this << '\0';
    strError = str();
    freeze(false);
    return;
}

string CLogMsg::getError()
{
    string strError;
    *this << '\0';
    strError = str();
    freeze(false);
    return strError;
}

void CLogMsg::getError(std::stringstream & strmError)
{
    *this << '\0';
    strmError << str();
    freeze(false);
    return;
}

////////////////////////////////////

const char CLogMsgEvent::bArgSep = '\t';

CLogMsgEvent::CLogMsgEvent()
{
    Init();
}

CLogMsgEvent::CLogMsgEvent(LPCSTR pszMessage)
    :CLogMsg(pszMessage)
{
    Init();
}

CLogMsgEvent::CLogMsgEvent(string & strMessage)
    :CLogMsg(strMessage)
{
    Init();
}

CLogMsgEvent::CLogMsgEvent(long lEventID, long lSeverity, LPCSTR pszMessage)
    :CLogMsg(pszMessage)
{
    Init();
    m_lEventID = lEventID;
    m_lSeverity = lSeverity;
}

CLogMsgEvent::CLogMsgEvent(long lEventID, long lSeverity, string & strMessage)
```

```
        :CLogMsg(strMessage)
    {
        Init();
        m_lEventID = lEventID;
        m_lSeverity = lSeverity;
    }

CLogMsgEvent::CLogMsgEvent(long lEventID, long lSeverity, __com_error & e)
{
    USES_CONVERSION;

    Init();
    m_lEventID = lEventID;
    m_lSeverity = lSeverity;

    *this << "0x" << std::hex << e.Error() << std::dec << bArgSep;
    BSTR bstrDesc = e.Description();
    if (bstrDesc != NULL)
        *this << W2T(bstrDesc);
    else
        *this << " ";
}

CLogMsgEvent::CLogMsgEvent(long lEventID, long lSeverity, HRESULT hr)
{
    Init();
    m_lEventID = lEventID;
    m_lSeverity = lSeverity;
    *this << "0x" << std::hex << hr;
}

CLogMsgEvent::~CLogMsgEvent()
{
    ReleaseBuffers();
}

inline void CLogMsgEvent::Init()
{
    m_lEventID = 0;
    m_lSeverity = -1;
    m_wArgCount = 0;
    m_ppszArgs = NULL;
}

void CLogMsgEvent::SetEvent(long lEventID, long lSeverity, LPCSTR pszMessage)
{
    Clear();
    m_lEventID = lEventID;
    m_lSeverity = lSeverity;
    if (pszMessage != NULL)
        *this << pszMessage;
}

long CLogMsgEvent::Event()
{
    return m_lEventID;
}

long CLogMsgEvent::Severity()
{
    if (m_lSeverity == -1)
    {
        if ((m_lEventID & 0xC0000000L) == 0xC0000000L)
            return EVENTLOG_ERROR_TYPE;
        else if (m_lEventID & 0x80000000L)
            return EVENTLOG_WARNING_TYPE;
        else if (m_lEventID & 0x40000000L)
            return EVENTLOG_INFORMATION_TYPE;
        else
            return EVENTLOG_SUCCESS;
    }
    return m_lSeverity;
}
```

```
        return EVENTLOG_INFORMATION_TYPE;
    else
        return EVENTLOG_SUCCESS;
}
else
    return m_lSeverity;
}
```

```
TCHAR ** CLogMsgEvent::Arguments(long * plArgCount)
{
```

```
    ReleaseBuffers();

    // get temp buffer
    strstream strmTemp;
    *this << '\0';
    strmTemp << str();
    freeze(false);

    // make sure double nulled
    strmTemp << '\0' << '\0';

    TCHAR * pszText = strmTemp.str();
    if (*pszText)
        m_wArgCount++;

    // make array of strings
    for (int i = 0; pszText[i]; i++)
    {
        if (pszText[i] == CLogMsgEvent::bArgSep)
        {
            pszText[i] = 0;
            m_wArgCount++;
        }
    }

    // if data, allocate arg array
    if (m_wArgCount)
    {
        int nLen = 0;
        m_ppszArgs = new TCHAR * [m_wArgCount];
        for (int i = 0; i < m_wArgCount; i++)
        {
            nLen = _tcslen(pszText);
            m_ppszArgs[i] = new TCHAR [nLen + 1];
            _tcscpy(m_ppszArgs[i], pszText);
            pszText += nLen + 1;
        }
    }

    strmTemp.freeze(false);

    // return buffer
    *plArgCount = m_wArgCount;
    return m_ppszArgs;
}
```

```
TCHAR * CLogMsgEvent::Text()
```

```
{
    CLogMsg::ReleaseBuffers();

    // format message into temporary strstream
    *this << '\0';
    std::strstream strmTemp;
    strmTemp << "Event:0x" << std::hex << m_lEventID << ", Severity:" << std::dec <<
    Severity() << ", Text:";

    // if translation is turned on, then get message from message source
```

```

    DWORD dwCharsReturned = 0;
    if (CLogNTEvents::m_hMsgSrc != NULL)
    {
        TCHAR pszBuff [2048];
        dwCharsReturned = FormatMessage(FORMAT_MESSAGE_FROM_HMODULE |
        FORMAT_MESSAGE_ARGUMENT_ARRAY,
        CLogNTEvents::m_hMsgSrc,
        m_lEventID,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        pszBuff,
        2048,
        m_ppszArgs);

        if (dwCharsReturned)
        {
            // chop off line feed
            pszBuff[--dwCharsReturned] = 0;
            // move data to formatted message
            if (dwCharsReturned)
                strmTemp << pszBuff << '\0';
        }
    }

    // if translation not turned on or translation didn't work then put out argument data
    if (!dwCharsReturned)
    {
        strmTemp << str() << '\0';
        freeze(false);
    }

    // move temp stringstream into m_pszText and return pointer to m_pszText
    int nLength = strmTemp.pcount();
    m_pszText = new TCHAR [nLength + 1];
    _tcsncpy(m_pszText, strmTemp.str(), nLength);
    strmTemp.freeze(false);
    m_pszText[nLength] = 0;

    return m_pszText;
}

void CLogMsgEvent::ReleaseBuffers()
{
    CLogMsg::ReleaseBuffers();

    if (m_ppszArgs != NULL)
    {
        for (int i = 0; i < m_wArgCount; i++)
            delete [] m_ppszArgs[i];
        delete [] m_ppszArgs;
        m_ppszArgs = NULL;
        m_wArgCount = 0;
    }

    return;
}

////////////////////////////////////
////////////////////////////////////
// Logs
////////////////////////////////////
////////////////////////////////////

CLogBase::CLogBase()
{
    m_fEnabled = true;
    m_nIndent = 0;
}

```



```
CLogBase::CLogBase(LPCSTR pszResourceName)
{
    m_fEnabled = true;
    m_strResourceName = pszResourceName;
    m_nIndent = 0;
}

void CLogBase::ResourceName(LPCSTR pszResourceName)
{
    m_strResourceName = pszResourceName;
    return;
}

void CLogBase::Post(CLogMsg * pmsgLog)
{
    return;
}

void CLogBase::Open()
{
    return;
}

void CLogBase::Close()
{
    return;
}

////////////////////////////////////
HINSTANCE CLogNTEvents::m_hMsgSrc = NULL;

CLogNTEvents::CLogNTEvents()
    :CLogBase()
{
}

CLogNTEvents::CLogNTEvents(LPCSTR pszResourceName)
    :CLogBase(pszResourceName)
{
}

void CLogNTEvents::Post(CLogMsg * pmsgLog)
{
    if (!m_fEnabled)
        return;

    HANDLE hEventSource = RegisterEventSource(NULL, m_strResourceName.c_str());
    if (hEventSource != NULL)
    {
        long lArgCount;
        TCHAR ** pszArgs = pmsgLog->Arguments(&lArgCount);
        ReportEvent(hEventSource, pmsgLog->Severity(), 0, pmsgLog->Event(), NULL,
lArgCount,
        0, (const TCHAR *) pszArgs, NULL);
        DeregisterEventSource(hEventSource);
    }
}

void CLogNTEvents::EnableTranslation(bool fEnable)
{
    if (fEnable)
    {
        if (!m_hMsgSrc)
            m_hMsgSrc = LoadMessageSource();
    }
    else
}
```

```
{
    if (m_hMsgSrc)
    {
        FreeLibrary(m_hMsgSrc);
        m_hMsgSrc = NULL;
    }
}

return;
}

HINSTANCE CLogNTEvents::LoadMessageSource()
{
    CRegistry    regLocal;

    // get the name of the resource
    if (!regLocal.Connect(CRegistry::keyLocalMachine))
        return NULL;

    string strKey("SYSTEM\\CurrentControlSet\\Services\\EventLog\\Application\\");
    strKey += m_strResourceName;
    if (!regLocal.Open(strKey.c_str()))
        return NULL;

    string strDLL;
    if (!regLocal.GetValue("EventMessageFile", strDLL))
        return NULL;

    // load the library
    return LoadLibrary(strDLL.c_str());
}

////////////////////////////////////

CLogFile::CLogFile()
    :CLogBase()
{
}

CLogFile::CLogFile(LPCSTR pszResourceName)
    :CLogBase(pszResourceName)
{
    m_streamIO.open(pszResourceName, ios_base::out | ios_base::trunc);
}

void CLogFile::Open(LPCSTR pszFileName)
{
    Close();
    m_strResourceName = pszFileName;
    Open();
}

void CLogFile::Open()
{
    if (!m_streamIO.is_open())
        m_streamIO.open(m_strResourceName.c_str(), ios_base::out | ios_base::trunc);
}

void CLogFile::Close()
{
    if (m_streamIO.is_open())
        m_streamIO.close();
    return;
}

void CLogFile::Post(CLogMsg * pmsgLog)
```

```
{
    if (!m_fEnabled)
        return;

    Lock();
    if (m_streamIO.is_open())
    {
        string strTabs(m_nIndent, '\\t');
        m_streamIO << strTabs << pmsgLog->Text() << '\\n';
        m_streamIO.flush();
    }
    Unlock();

    return;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

CLogDebug::CLogDebug()
{
}

void CLogDebug::Post(CLogMsg * pmsgLog)
{
    if (!m_fEnabled)
        return;
    if (m_nIndent)
    {
        string strTabs(m_nIndent, '\\t');
        OutputDebugString(strTabs.c_str());
    }
    OutputDebugString(pmsgLog->Text());
    OutputDebugString("\\n");
    return;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

CLogMulti::CLogMulti()
{
}

void CLogMulti::AddLog(CLogBase * plog)
{
    Lock();
    m_collLogs.push_back(plog);
    Unlock();
}

void CLogMulti::RemoveLog(CLogBase * plog)
{
    Lock();
    if (plog != NULL)
        m_collLogs.remove(plog);
    else
        m_collLogs.erase(m_collLogs.begin(), m_collLogs.end());
    Unlock();
    return;
}

void CLogMulti::Post(CLogMsg * pmsgLog)
{
    if (!m_fEnabled)
        return;

    Lock();
    list<CLogBase *>::iterator itLogs;
```

```
    for (itLogs = m_collLogs.begin(); itLogs != m_collLogs.end(); itLogs++)
        pmsgLog->Post>(*itLogs);
    Unlock();

    return;
}

void CLogMulti::Open()
{
    Lock();
    list<CLogBase *>::iterator itLogs;
    for (itLogs = m_collLogs.begin(); itLogs != m_collLogs.end(); itLogs++)
        (*itLogs)->Open();
    Unlock();
}

void CLogMulti::Close()
{
    Lock();
    list<CLogBase *>::iterator itLogs;
    for (itLogs = m_collLogs.begin(); itLogs != m_collLogs.end(); itLogs++)
        (*itLogs)->Close();
    Unlock();
}

string CTimeStamp::LocalTime()
{
    SYSTEMTIME tm;
    GetLocalTime(&tm);
    char pBuff [30];
    sprintf(pBuff, "%02d:%02d:%02d.%d", tm.wHour, tm.wMinute, tm.wSecond, tm.
        wMilliseconds);
    return string(pBuff);
}
```

```
#pragma once
```

```
/* This code was taken from the "Windows Foundation Class" project  
which is authored by Samuel R. Blackburn (see the below original  
comments from Sam.)
```

```
The source code used MFC as a basis but since the AHC source  
code avoids MFC, I have modified this code to use nothing but  
standard C++. Also, I've have removed functionality that did  
not make sense in the AHC case to lessen the amount of code  
present.
```

```
Darin Greaham  
Millbrook Corporation  
August 1997
```

```
*/
```

```
/*  
** Author: Samuel R. Blackburn  
** CIs: 76300,326  
** Internet: sblackbu@erols.com  
**  
** You can use it any way you like as long as you don't try to sell it.  
**  
** Any attempt to sell WFC in source code form must have the permission  
** of the original author. You can produce commercial executables with  
** WFC but you can't sell WFC.  
**  
** Copyright, 1997, Samuel R. Blackburn  
**  
** $Workfile: Registry.h $  
** $Revision: 1 $  
** $Modtime: 1/14/00 1:49p $  
*/
```

```
class CRegistry
```

```
{  
public:  
    CRegistry( const CRegistry& RightSide )  
    {  
        // call assignment operator  
        *this = RightSide;  
    }  
  
    CRegistry& operator=( const CRegistry& RightSide )  
    {  
        m_KeyHandle = RightSide.m_KeyHandle;  
        m_RegistryHandle = RightSide.m_RegistryHandle;  
        m_ErrorCode = RightSide.m_ErrorCode;  
        m_ClassName = RightSide.m_ClassName;  
        m_ComputerName = RightSide.m_ComputerName;  
        m_KeyName = RightSide.m_KeyName;  
        m_RegistryName = RightSide.m_RegistryName;  
        m_NumberOfSubkeys = RightSide.m_NumberOfSubkeys;  
        m_NumberOfValues = RightSide.m_NumberOfValues;  
        m_LongestSubkeyNameLength = RightSide.m_LongestSubkeyNameLength;  
        m_LongestClassNameLength = RightSide.m_LongestClassNameLength;  
        m_LongestValueNameLength = RightSide.m_LongestValueNameLength;  
        m_LongestValueDataLength = RightSide.m_LongestValueDataLength;  
        m_SecurityDescriptorLength = RightSide.m_SecurityDescriptorLength;  
        m_LastWriteTime = RightSide.m_LastWriteTime;  
        return( *this );  
    };  
  
private:  
    void m_Initialize( void );
```

protected:

HKEY m_KeyHandle;
HKEY m_RegistryHandle;

LONG m_ErrorCode;

string m_ClassName;
string m_ComputerName;
string m_KeyName;
string m_RegistryName;
DWORD m_NumberOfSubkeys;
DWORD m_NumberOfValues;

//
// Data items filled in by QueryInfo
//

DWORD m_LongestSubkeyNameLength;
DWORD m_LongestClassNameLength;
DWORD m_LongestValueNameLength;
DWORD m_LongestValueDataLength;
DWORD m_SecurityDescriptorLength;
FILETIME m_LastWriteTime;

public:

enum _Keys

{
keyLocalMachine = (DWORD) HKEY_LOCAL_MACHINE,
keyClassesRoot = (DWORD) HKEY_CLASSES_ROOT,
keyPerformanceData = (DWORD) HKEY_PERFORMANCE_DATA,
keyUsers = (DWORD) HKEY_USERS,
keyCurrentUser = (DWORD) HKEY_CURRENT_USER,
#if (WINVER >= 0x400)
keyCurrentConfiguration = (DWORD) HKEY_CURRENT_CONFIG,
keyDynamicData = (DWORD) HKEY_DYNAMIC_DATA
#endif

#endif
};

enum KeyValueTypes

{
typeBinary = REG_BINARY,
typeDoubleWord = REG_DWORD,
typeDoubleWordLittleEndian = REG_DWORD_LITTLE_ENDIAN,
typeDoubleWordBigEndian = REG_DWORD_BIG_ENDIAN,
typeUnexpandedString = REG_EXPAND_SZ,
typeSymbolicLink = REG_LINK,
typeMultipleString = REG_MULTI_SZ,
typeNone = REG_NONE,
typeResourceList = REG_RESOURCE_LIST,
typeString = REG_SZ
};

enum CreateOptions

{
optionsNonVolatile = REG_OPTION_NON_VOLATILE,
optionsVolatile = REG_OPTION_VOLATILE
};

enum CreatePermissions

{
permissionAllAccess = KEY_ALL_ACCESS,
permissionCreateLink = KEY_CREATE_LINK,
permissionCreateSubKey = KEY_CREATE_SUB_KEY,
permissionEnumerateSubKeys = KEY_ENUMERATE_SUB_KEYS,
permissionExecute = KEY_EXECUTE,

```

    permissionNotify          = KEY_NOTIFY,
    permissionQueryValue      = KEY_QUERY_VALUE,
    permissionRead            = KEY_READ,
    permissionSetValue        = KEY_SET_VALUE,
    permissionWrite           = KEY_WRITE
};

enum CreationDisposition
{
    dispositionCreatedNewKey    = REG_CREATED_NEW_KEY,
    dispositionOpenedExistingKey = REG_OPENED_EXISTING_KEY
};

CRegistry();

//
// Destructor should be virtual according to MSJ article in Sept 1992
// "Do More with Less Code:..."
//

virtual ~CRRegistry();

//
// Methods
//

virtual BOOL Close( void );

virtual BOOL Connect( const _Keys key_to_open = keyCurrentUser,
                     LPCTSTR computer_name = NULL );

virtual BOOL Create( LPCTSTR name_of_subkey,
                    LPCTSTR name_of_class = NULL,
                    CreateOptions options = NULL,
optionsNonVolatile, CreatePermissions permissions =
permissionAllAccess, LPSECURITY_ATTRIBUTES security_attributes_p = NULL,
                    CreationDisposition * disposition_p = NULL );

virtual BOOL DeleteKey( LPCTSTR name_of_subkey_to_delete );

virtual BOOL DeleteValue( LPCTSTR name_of_value_to_delete );

virtual BOOL EnumerateKeys( const DWORD subkey_index,
                           string& subkey_name,
                           string& class_name );

virtual BOOL EnumerateValues( const DWORD value_index,
                              string& name_of_value,
                              KeyValueTypes& type_code,
                              LPBYTE data_buffer,
                              DWORD& size_of_data_buffer );

virtual BOOL Flush( void );
virtual BOOL GetBinaryValue( LPCTSTR name_of_value, BYTE return_array[], DWORD&
num_bytes_read );
virtual void GetClassName( string& class_name ) const;
virtual void GetComputerName( string& computer_name ) const;
virtual BOOL GetDoubleWordValue( LPCTSTR name_of_value, DWORD& return_value );
virtual BOOL GetErrorCode( void ) const;
virtual void GetKeyName( string& key_name ) const;
virtual DWORD GetNumberOfSubkeys( void ) const;
virtual DWORD GetNumberOfValues( void ) const;
virtual void GetRegistryName( string& registry_name ) const;
virtual BOOL GetStringValue( LPCTSTR name_of_value, string& return_string );

```



```
#ifndef registryBase h
#define _registryBase_h

#include "registry.h"

class CRegistryBase
{
protected:
    CRegistry          m_oRegistry;

    enum {REG_FAILURE = 0, REG_SUCCESS = 1};

    CRegistryBase();
    bool connect(CRegistry::Keys eKey, LPCSTR pszComputer = NULL);
    bool getValue(LPCSTR pszValueName, string & strValue);
    bool getValue(LPCSTR pszValueName, unsigned long & lValue);
    bool getBinaryValue(LPCSTR pszValueName, LPBYTE pBuff, DWORD * pdwBuffSize);
    bool setValue(LPCSTR pszValueName, LPCSTR pszValue);
    bool setValue(LPCSTR pszValueName, unsigned long lValue);
    bool openKey(LPCSTR pszKeyPath);
    bool createKey(LPCSTR pszKeyPath);
    int enumKeys(vector<string> & aryKeys);
    bool close() { return (m_oRegistry.Close() == TRUE); }

public:
    string          m_strLastError;
};

#endif
```

```
#ifndef registryDB h
#define _registryDB_h

#include "registryBase.h"

class CRegistryDatabase : public CRegistryBase
{
protected:
    static string          m_strPath;
    bool                   m_fProductOpen;
    bool                   m_fAliasOpen;

public:
    vector<string>          m_collAliases;
    string                 m_strProduct;
    string                 m_strAlias;
    string                 m_strDefaultAlias;
    string                 m_strDatabaseName;
    string                 m_strDatabaseType;
    string                 m_strPassword;
    string                 m_strServerName;
    string                 m_strUserName;
    string                 m_strDBType;

protected:
    void Init();

public:
    CRegistryDatabase();
    bool openProduct(LPCSTR pszProduct);
    bool openAlias(LPCSTR pszAlias);
};

#endif
```

```
//{{NO DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by LCBroker.rc
//
#define IDS_SERVICENAME 100
#define IDR_LCBroker 100
#define IDR_LCBROKER1 101

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifdef APSTUDIO_READONLY_SYMBOLS
#define APS_NEXT_RESOURCE_VALUE 201
#define APS_NEXT_COMMAND_VALUE 32768
#define APS_NEXT_CONTROL_VALUE 201
#define _APS_NEXT_SYMED_VALUE 102
#endif
#endif
```

```
#include "stdafx.h"
#define _rs_cpp

#include "rs_address.cpp"
#include "rs_admit.cpp"
#include "rs_allergy.cpp"
#include "rs_audit.cpp"
#include "rs_convert_pc.cpp"
#include "rs_discharge.cpp"
#include "rs_employers.cpp"
#include "rs_encounter_tree.cpp"
#include "rs_encounter.cpp"
#include "rs_family_tree.cpp"
#include "rs_id_map.cpp"
#include "rs_insurance.cpp"
#include "rs_loa.cpp"
#include "rs_misc_id.cpp"
#include "rs_name.cpp"
#include "rs_name_search.cpp"
#include "rs_person.cpp"
#include "rs_phone.cpp"
#include "rs_physical.cpp"
#include "rs_pre_admit.cpp"
#include "rs_transfer.cpp"
#include "rs_code_cache.cpp"
#include "rs_cpi_user.cpp"
#include "rs_account.cpp"
#include "rs_disability.cpp"
#include "rs_care_directives.cpp"
#include "rs_guarantor.cpp"
#include "rs_diagnosis.cpp"
#include "rs_physicians.cpp"
#include "rs_patient_valuables.cpp"
#include "rs_company.cpp"
#include "rs_decision.cpp"
#include "rs_hcp.cpp"
#include "rs_sys_org_facility.cpp"
#include "rs_cpi_master.cpp"
#include "rs_nok.cpp"
#include "rs_location.cpp"
#include "rs_patient.cpp"
#include "rs_blood_pressure.cpp"
#include "rs_health_condition.cpp"
#include "rs_immunization.cpp"
#include "rs_medication.cpp"
#include "rs_surgery.cpp"
#include "rs_therapy.cpp"
#include "rs_family_history.cpp"
#include "rs_imaging.cpp"
#include "rs_reminders.cpp"
#include "rs_cholesterol.cpp"
#include "rs_mass_mailing.cpp"
#include "rs_unregistered_user.cpp"
#include "rs_stats.cpp"
#include "rs_user_preference.cpp"
#include "rs_kiosk.cpp"
```

```
// stdafx.h : include file for standard system include files,
//          or project specific include files that are used frequently,
//          but are changed infrequently

#if !defined(AFX_STDAFX_H__75751D74_AFD1_11D2_AC59_00C04F6E4C48__INCLUDED_)
#define AFX_STDAFX_H__75751D74_AFD1_11D2_AC59_00C04F6E4C48__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define STRICT
#ifndef WIN32_WINNT
#define _WIN32_WINNT 0x0400
#endif
// #define _ATL_APARTMENT_THREADED

//include for using COleDateTime class
#include <afxdisp.h>

#include <atlbase.h>
//You may derive a class from CComModule and use it if you want to override
//something, but do not change the name of _Module

struct CLCBrokerStats
{
    SYSTEMTIME      m_sysTimeStarted;
    DATE            m_vartimeStarted;
    long            m_lCommandsProcessed;
    long            m_lTotalClients;
    long            m_lCurrentClients;

    CLCBrokerStats()
    {
        memset(&m_sysTimeStarted, 0x00, sizeof(SYSTEMTIME));
        m_vartimeStarted = 0.0f;
        m_lCommandsProcessed = 0;
        m_lTotalClients = 0;
        m_lCurrentClients = 0;
    }
};

//STL
#include <string>

class CServiceModule : public CComModule
{
public:
    HRESULT RegisterServer(BOOL bRegTypeLib, BOOL bService);
    HRESULT UnregisterServer();
    void Init(_ATL_OBJMAP_ENTRY* p, HINSTANCE h, UINT nServiceNameID, const GUID* plibid =
        NULL);
    void Start();
    void ServiceMain(DWORD dwArgc, LPTSTR* lpszArgv);
    void Handler(DWORD dwOpcode);
    void Run();
    BOOL IsInstalled();
    BOOL Install();
    BOOL Uninstall();
    LONG Unlock();
    void LogEvent(LPCTSTR pszFormat, ...);
    void SetServiceStatus(DWORD dwState);
    void SetupAsLocalServer();

//Implementation
private:
```

```

    static void WINAPI ServiceMain(DWORD dwArgc, LPTSTR* lpszArgv);
    static void WINAPI _Handler(DWORD dwOpcode);

// data members
public:
    TCHAR m_szServiceName[256];
    SERVICE_STATUS_HANDLE m_hServiceStatus;
    SERVICE_STATUS m_status;
    DWORD dwThreadId;
    BOOL m_bService;
    CLCBrokerStats m_statsLCBroker;

};

extern CServiceModule _Module;
#include <atlcom.h>

////////////////////////////////////
// COM
#pragma warning(disable:4192)
#import "msxml.dll"
#import "msado15.dll" no_namespace rename("EOF", "EOFado")
#import "..\..\..\common\applications\idgenerator\tlb\idgenerator.tlb" no_namespace
#import "..\..\..\cpi\backend\cpisearcher\idloutput\cpisearcher.tlb" no_namespace

////////////////////////////////////
// STL
#include <locale>
#include <vector>
#include <map>
#include <sstream>
#include <fstream>
#include <list>
using namespace std;
#define TOUPPER(str) ctype<string::value_type>().toupper(str.begin(), str.end())
#define TOLOWER(str) ctype<string::value_type>().tolower(str.begin(), str.end())

////////////////////////////////////
// CRT
#include <assert.h>

////////////////////////////////////
// MS
#include <search.h>

////////////////////////////////////
// SLMD

// event and debug logging
#include "Encryptor.h"
#include "HL7EvMsgSrc.h"
#include "Logging.h"
extern CLogNTEvents      logEvents;
extern CLogFile          logFile;
extern CLogDebug         logDebug;
extern CLogMulti         logAll;
extern string            _strDefaultAlias;

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
// previous line.

#endif // !defined(AFX_STDAFX_H__75751D74_AFD1_11D2_AC59_00C04F6E4C48__INCLUDED)

```

```
#include "xc_OtherCommands.h"

#include "rs_insurance.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_addInsurance)

////////////////////////////////////////
//Execute the command. [Call execute and write processing code in processData()]
////////////////////////////////////////
bool Cxc_addInsurance::execCommand()
{
    return execute();
}

////////////////////////////////////////
//Do parameter validation here
////////////////////////////////////////
bool Cxc_addInsurance::parseParameters ()
{
    string strData;
    bool fPresent;

    //////////////////////////////////////////
    //cpi_id should be provided.
    //////////////////////////////////////////
    strData = getParameterValue("cpi_id");
    if (strData.empty())
    {
        m_emLast.setError("\cpi_id\" is a required.");
        return false;
    }

    //////////////////////////////////////////
    //Company name should be provided
    //////////////////////////////////////////
    strData = getParameterValue("company_name");
    if (strData.empty())
    {
        m_emLast.setError("\company_name\" is required.");
        return false;
    }

    //////////////////////////////////////////
    //Self Insured Swith should be provided.
    //////////////////////////////////////////
    fPresent = false;
    strData = getParameterValue("self_insured_sw");
    if (strData.empty())
    {
        m_emLast.setError("\self_insured_sw\" is required.");
        return false;
    }

    bool fSelfInsured = (strData == "1") ? true : false;

    //////////////////////////////////////////
    //Subscriber last_name should be provided if any name components are provided
    //////////////////////////////////////////
    fPresent = false;
    strData = getParameterValue("subscriber_last_name");
    if (strData.empty())
    {

```

```
        if (!getParameterValue("subscriber_first_name").empty()) fPresent = true;
        if (!getParameterValue("subscriber_middle_name").empty()) fPresent = true;

        if (fPresent)
        {
            m_emLast.setError("\\"Last Name\\" is required, if any other name components
are provided.");
            return false;
        }

        //No subscriber info provided, so make sure self_insured switch is "1"
        if (!fSelfInsured)
        {
            m_emLast.setError("No subscriber info provided for dependent !!!");
            return false;
        }
    }

    //////////////////////////////////////
    //Code ID's should be provided if any codes are provided
    //////////////////////////////////////
    if (!getParameterValue("state").empty() && getParameterValue("state_id").empty())
    {
        m_emLast.setError("\\"state_id\\" is not present. Codes should be accompanied by its
CodeID.");
        return false;
    }
    if (!getParameterValue("country").empty() && getParameterValue("country_id").empty())
    {
        m_emLast.setError("\\"country_id\\" is not present. Codes should be accompanied by
its CodeID.");
        return false;
    }
    if (!getParameterValue("claims_state").empty() && getParameterValue("claims_state_id").
empty())
    {
        m_emLast.setError("\\"claims_state_id\\" is not present. Codes should be accompanied
by its CodeID.");
        return false;
    }
    if (!getParameterValue("claims_country").empty() && getParameterValue(
"claims_country_id").empty())
    {
        m_emLast.setError("\\"claims_country_id\\" is not present. Codes should be
accompanied by its CodeID.");
        return false;
    }
    if (!getParameterValue("subscriber_relationship").empty() && getParameterValue(
"subscriber_relationship_id").empty())
    {
        m_emLast.setError("\\"subscriber_relationship_id\\" is not present. Codes should be
accompanied by its CodeID.");
        return false;
    }

    return true;
}

////////////////////////////////////
// Do Data processing here.
// [called from the execute method for each row of data]
//
// - creates insurance company, subscriber and participant insurance records.
```



```
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
bool Cxc_addInsurance::processData ()
{
    bool fSuccess = true;
    CSdoConnection * pconn = NULL;

    try
    {
        Crs_name          rs_name;
        Crs_address       rs_address;
        Crs_phone         rs_phone;
        Crs_cpi_master    rs_cpi_master;
        Crs_insurance     rs_insurance;
        Crs_company       rs_company;

        string strCompanyName, strStreet1, strStreet2, strCity, strZip, strState,
        strCountry;
        string strClaimsStreet1, strClaimsStreet2, strClaimsCity, strClaimsState,
        strClaimsZip, strClaimsCountry;
        string strPhoneEmergency, strPhoneMentalHealth, strPhonePreCert, strPhoneBenefits,
        strPhoneOther;
        string strPlanCode, strPlanType, strPlanEffectiveDate, strPolicyNumber,
        strGroupNumber, strGroupName;
        string strSubLastName, strSubFirstName, strSubMiddleName, strSubRelationship,
        strSubPhone;

        long lCpiId, lStateId, lCountryId, lClaimsStateId, lClaimsCountryId,
        lSubRelationshipId;
        bool fSelfInsured;
        long lCompanyId;

        //get the parameters
        lCpiId = atol(getParameterValue("cpi_id").c_str());
        lCompanyId = atol(getParameterValue("company_id").c_str());
        strCompanyName = getParameterValue("company_name");
        strStreet1 = getParameterValue("street1");
        strStreet2 = getParameterValue("street2");
        strCity = getParameterValue("city");
        strState = getParameterValue("state");
        lStateId = atol(getParameterValue("state_id").c_str());
        strZip = getParameterValue("zip");
        strCountry = getParameterValue("country");
        lCountryId = atol(getParameterValue("country_id").c_str());
        strClaimsStreet1 = getParameterValue("claims_street1");
        strClaimsStreet2 = getParameterValue("claims_street2");
        strClaimsCity = getParameterValue("claims_city");
        strClaimsState = getParameterValue("claims_state");
        lClaimsStateId = atol(getParameterValue("claims_state_id").c_str());
        strClaimsZip = getParameterValue("claims_zip");
        strClaimsCountry = getParameterValue("claims_country");
        lClaimsCountryId = atol(getParameterValue("claims_country_id").c_str());
        strPhoneEmergency = getParameterValue("phone_emergency");
        strPhoneMentalHealth = getParameterValue("phone_mental_health");
        strPhonePreCert = getParameterValue("phone_precert");
        strPhoneBenefits = getParameterValue("phone_benefits");
        strPhoneOther = getParameterValue("phone_other");
        strPlanCode = getParameterValue("plan_code");
        strPlanType = getParameterValue("plan_type");
        strPlanEffectiveDate = getParameterValue("plan_effective_dt");
        strPolicyNumber = getParameterValue("policy_number");
        strGroupNumber = getParameterValue("group_number");
        strGroupName = getParameterValue("group_name");
        strSubLastName = getParameterValue("subscriber_last_name");
```

```

    strSubFirstName = getParameterValue("subscriber_first_name");
    strSubMiddleName = getParameterValue("subscriber_middle_name");
    strSubRelationship = getParameterValue("subscriber_relationship");
    lSubRelationshipId = atol(getParameterValue("subscriber_relationship_id").c_str()) ✓

;

    strSubPhone = getParameterValue("subscriber_phone");
    fSelfInsured = (getParameterValue("self_insured_sw") == "1") ? true : false;

    //convert date in date time object.
    DATE dtPlanEffectiveDate;
    COleDateTime oledate;

    oledate.ParseDateTime(strPlanEffectiveDate.c_str());
    dtPlanEffectiveDate = (DATE) oledate;

    if (!dtPlanEffectiveDate)
    {
        //return error if date is made compulsory.
    }

    //get db connection.
    pconn = m_pcoClient->getConnection();

    //begin transaction
    pconn->beginTrans();

    //get new audit id
    long lAuditId = getAuditId();
    if (!lAuditId)
    {
        m_emLast.setError("Unexpected Condition !!! Cannot get new Audit ID !!!");
        throw fSuccess = false;
    }

    /*****
    // COMPANY: Create company record.
    *****/

    char szBuffer[20];

    //create company record only if company_id not provided.
    if (!lCompanyCpiId)
    {
        //fetch new cpi_id
        lCompanyCpiId = getNewCpiId();
        if (!lCompanyCpiId)
        {
            m_emLast.setError("Unexpected condition!!! Cannot get new cpi id for ✓
company !!!");
            throw fSuccess = false;
        }
    }

    string strCompanyCpiId = "cpi";
    strCompanyCpiId += ltoa(lCompanyCpiId, szBuffer, 10);

    //insert new record in cpi_master
    rs_cpi_master.clearParms();
    rs_cpi_master.setRecordSetToNull();
    rs_cpi_master.setActiveCommand("cmdInsertEmptyRecord");
    rs_cpi_master.setParameter("cpi_id", _variant_t (lCompanyCpiId));
    rs_cpi_master.setParameter("cpi_text_id", _variant_t (strCompanyCpiId.c_str ✓
    ));
    rs_cpi_master.setParameter("audit_id", _variant_t (lAuditId));
    if (!pconn->execute(rs_cpi_master))
    {
        m_emLast.setError(pconn->getLastError());
    }

```

```

        throw fSuccess = false;
    }

    //create company record in company table
    rs_company.clearParms();
    rs_company.setRecordSetToNull();
    rs_company.setActiveCommand("cmdUpdate");
    rs_company.setParameter("cpi_id", _variant_t (lCompanyCpiId));
    rs_company.setParameter("name", _variant_t (strCompanyName.c_str()));
    rs_company.setParameter("audit_id", _variant_t (lAuditId));
    if ((fSuccess = pconn->execute(rs_company)) == false)
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }

    //create company address record
    rs_address.clearParms();
    rs_address.setRecordSetToNull();
    rs_address.setActiveCommand("cmdUpdate");
    rs_address.setParameter("cpi_id", _variant_t (lCompanyCpiId));
    rs_address.setParameter("active_sw", _variant_t ("1"));
    rs_address.setParameter("primary_sw", _variant_t ("1"));
    rs_address.setParameter("purpose", _variant_t ("Work"));
    rs_address.setParameter("street1", _variant_t (strStreet1.c_str()));
    rs_address.setParameter("street2", _variant_t (strStreet2.c_str()));
    rs_address.setParameter("city", _variant_t (strCity.c_str()));
    rs_address.setParameter("state", _variant_t (strState.c_str()));
    rs_address.setParameter("zip", _variant_t (strZip.c_str()));
    rs_address.setParameter("country", _variant_t (strCountry.c_str()));
    rs_address.setParameter("audit_id", _variant_t (lAuditId));
    if (lStateId) rs_address.setParameter("state_id", _variant_t (lStateId));
    if (lCountryId) rs_address.setParameter("country_id", _variant_t (lCountryId));

    if ((fSuccess = pconn->execute(rs_address)) == false)
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }

    //create company claims address record
    rs_address.clearParms();
    rs_address.setRecordSetToNull();
    rs_address.setActiveCommand("cmdUpdate");
    rs_address.setParameter("cpi_id", _variant_t (lCompanyCpiId));
    rs_address.setParameter("active_sw", _variant_t ("1"));
    rs_address.setParameter("purpose", _variant_t ("Claims"));
    rs_address.setParameter("street1", _variant_t (strClaimsStreet1.c_str()));
    rs_address.setParameter("street2", _variant_t (strClaimsStreet2.c_str()));
    rs_address.setParameter("city", _variant_t (strClaimsCity.c_str()));
    rs_address.setParameter("state", _variant_t (strClaimsState.c_str()));
    rs_address.setParameter("zip", _variant_t (strClaimsZip.c_str()));
    rs_address.setParameter("country", _variant_t (strClaimsCountry.c_str()));
    rs_address.setParameter("audit_id", _variant_t (lAuditId));
    if (lClaimsStateId) rs_address.setParameter("state_id", _variant_t
(lClaimsStateId));
    if (lClaimsCountryId) rs_address.setParameter("country_id", _variant_t
(lClaimsCountryId));
    if ((fSuccess = pconn->execute(rs_address)) == false)
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }

    //create company phone records

```

```
//create emergency room phone
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdUpdate");
rs_phone.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_phone.setParameter("number", _variant_t (strPhoneEmergency.c_str()));
rs_phone.setParameter("purpose", _variant_t ("EROOM"));
rs_phone.setParameter("active_sw", _variant_t ("1"));
rs_phone.setParameter("audit_id", _variant_t (lAuditId));
if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//create mental health phone
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdUpdate");
rs_phone.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_phone.setParameter("number", _variant_t (strPhoneMentalHealth.c_str()));
rs_phone.setParameter("purpose", _variant_t ("MHEALTH"));
rs_phone.setParameter("active_sw", _variant_t ("1"));
rs_phone.setParameter("audit_id", _variant_t (lAuditId));
if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//create Pre Certification phone
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdUpdate");
rs_phone.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_phone.setParameter("number", _variant_t (strPhonePreCert.c_str()));
rs_phone.setParameter("purpose", _variant_t ("PRECERT"));
rs_phone.setParameter("active_sw", _variant_t ("1"));
rs_phone.setParameter("audit_id", _variant_t (lAuditId));
if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//create Benefits phone
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdUpdate");
rs_phone.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_phone.setParameter("number", _variant_t (strPhoneBenefits.c_str()));
rs_phone.setParameter("purpose", _variant_t ("BENEFITS"));
rs_phone.setParameter("active_sw", _variant_t ("1"));
rs_phone.setParameter("audit_id", _variant_t (lAuditId));
if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//create Other phone
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdUpdate");
rs_phone.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_phone.setParameter("number", _variant_t (strPhoneOther.c_str()));
rs_phone.setParameter("purpose", _variant_t ("OTHER"));
```

```

        rs_phone.setParameter("active_sw", _variant_t ("1"));
        rs_phone.setParameter("audit_id", _variant_t (lAuditId));
        if ((fSuccess = pconn->execute(rs_phone)) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
    }

    /*****
    // SUBSCRIBER: Create subscriber record/update user record if self insured
    *****/

    long lSubCpiId = 0;
    long lNameRecId = 0;
    long lPhoneRecId = 0;

    //create subscriber if not self insured.
    if (!fSelfInsured)
    {
        //fetch new cpi_id
        lSubCpiId = getNewCpiId();
        if (!lSubCpiId)
        {
            m_emLast.setError("Unexpected condition!!! Cannot get new cpi id for
subscriber !!!");
            throw fSuccess = false;
        }

        string strSubCpiId = "cpi";
        strSubCpiId += ltoa(lSubCpiId, szBuffer, 10);

        //insert new record in cpi_master
        rs_cpi_master.clearParms();
        rs_cpi_master.setRecordSetToNull();
        rs_cpi_master.setActiveCommand("cmdInsertEmptyRecord");
        rs_cpi_master.setParameter("cpi_id", _variant_t (lSubCpiId));
        rs_cpi_master.setParameter("cpi_text_id", _variant_t (strSubCpiId.c_str()));
        rs_cpi_master.setParameter("audit_id", _variant_t (lAuditId));
        if (!pconn->execute(rs_cpi_master))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }

        //The subscriber name and phone records has to be created
        //so nullify the record id's so it will create new records.
        lNameRecId = 0;
        lPhoneRecId = 0;
    }
    else
    {
        //user is self insured, so subscriber info is actually the user info.
        //so update the user info passed under the subscriber fields

        string strRecId;

        //user is the subscriber as he is self insured.
        lSubCpiId = lCpiId;

        //Get the record id's so the user information is updated
    }

```

```
//get rec_id for user name record for updation
lNameRecId = 0;
rs_name.clearParms();
rs_name.setRecordSetToNull();
rs_name.setActiveCommand("cmdFetchRecordId");
rs_name.setParameter("cpi_id", _variant_t (lSubCpiId));
if ((fSuccess = pconn->execute(rs_name)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
if (!rs_name.isEmpty())
{
    rs_name.getField("rec_id", strRecId);
    lNameRecId = atol(strRecId.c_str());
    rs_name.setRecordSetToNull();
}

//get rec_id for user phone record for updation
lPhoneRecId = 0;
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdFetchRecordIdByPurpose");
rs_phone.setParameter("cpi_id", _variant_t (lSubCpiId));
rs_phone.setParameter("purpose", _variant_t ("Home"));
rs_phone.setParameter("line_type", _variant_t ("Voice"));
if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
if (!rs_phone.isEmpty())
{
    rs_phone.getField("rec_id", strRecId);
    lPhoneRecId = atol(strRecId.c_str());
    rs_phone.setRecordSetToNull();
}

}

//update subscriber/user name record
if (!strSubLastName.empty())
{
    rs_name.clearParms();
    rs_name.setRecordSetToNull();
    rs_name.setActiveCommand("cmdUpdate");
    rs_name.setParameter("cpi_id", _variant_t (lSubCpiId));
    rs_name.setParameter("active_sw", _variant_t ("1"));
    rs_name.setParameter("last_name", _variant_t (strSubLastName.c_str()));
    rs_name.setParameter("middle_name", _variant_t (strSubMiddleName.c_str()));
    rs_name.setParameter("first_name", _variant_t (strSubFirstName.c_str()));
    rs_name.setParameter("audit_id", _variant_t (lAuditId));

    if (lNameRecId)
        rs_name.setParameter("rec_id", _variant_t (lNameRecId));

    if ((fSuccess = pconn->execute(rs_name)) == false)
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }
}

//update subscriber/user HOME phone record
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
```

```

rs_phone.setActiveCommand("cmdUpdate");
rs_phone.setParameter("cpi_id", _variant_t (lSubCpiId));
rs_phone.setParameter("number", _variant_t (strSubPhone.c_str()));
rs_phone.setParameter("active_sw", _variant_t ("1"));
rs_phone.setParameter("audit_id", _variant_t (lAuditId));

if (lPhoneRecId)
    rs_phone.setParameter("rec_id", _variant_t (lPhoneRecId));

if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

/*****
// INSURANCE: Create insurance record.
*****/

//create insurance records.
rs_insurance.clearParms();
rs_insurance.setRecordSetToNull();
rs_insurance.setActiveCommand("cmdUpdate");
rs_insurance.setParameter("cpi_id", _variant_t (lCpiId));
rs_insurance.setParameter("active_sw", _variant_t ("1"));
rs_insurance.setParameter("ins_co_id", _variant_t (lCompanyCpiId));
rs_insurance.setParameter("insured_id", _variant_t (lSubCpiId));
rs_insurance.setParameter("group_name", _variant_t (strGroupName.c_str()));
rs_insurance.setParameter("group_number", _variant_t (strGroupNumber.c_str()));
rs_insurance.setParameter("policy_number", _variant_t (strPolicyNumber.c_str()));
rs_insurance.setParameter("ins_plan_code", _variant_t (strPlanCode.c_str()));
rs_insurance.setParameter("plan_type_code", _variant_t (strPlanType.c_str()));
if (dtPlanEffectiveDate)
    rs_insurance.setParameter("plan_eff_dt", _variant_t (dtPlanEffectiveDate));
rs_insurance.setParameter("insured_relationship", _variant_t (strSubRelationship.
c_str()));
rs_insurance.setParameter("audit_id", _variant_t (lAuditId));
if (lSubRelationshipId)
    rs_insurance.setParameter("insured_relationship_id", _variant_t
(lSubRelationshipId));

if ((fSuccess = pconn->execute(rs_insurance)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

}
catch(bool fError)
{
    fError;
}
catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}
catch(...)
{
    m_emLast.setError("Unknown exception raised. [Command:addInsurance]");
    fSuccess = false;
}

```

```
//commit or Roll back the transaction.
if (pconn)
{
    if (fSuccess)    pconn->commitTrans();
    else             pconn->rollbackTrans();
}

return fSuccess;
}
```



```
#include "stdafx.h"
#include "LCBroker.h"
#include "CoLCBroker.h"
#include "xcLCBroker.h"
#define _xc_cpp

////////////////////////////////////
// compile unit for cpp commands

#include "xcLCBroker.cpp"
#include "xcLCBrokerModify.cpp"

//General Commands include
#include "xc openDatabase.cpp"
#include "xc loginUser.cpp"
#include "xc execSearch.cpp"
#include "xc createUser.cpp"
#include "xc_changePassword.cpp"
#include "xc_setBloodPressure.cpp"
#include "xc_setUserBiographics.cpp"
#include "xc_setSLMDLocations.cpp"
#include "xc_setEmploymentInfo.cpp"
#include "xc_setUserPhysicians.cpp"
#include "xc_addInsurance.cpp"
#include "xc_setInsurance.cpp"
#include "xc_setAllergyInfo.cpp"
#include "xc_setHealthConditions.cpp"
#include "xc_setImmunizations.cpp"
#include "xc_setMedications.cpp"
#include "xc_setSurgeryInfo.cpp"
#include "xc_setTherapyInfo.cpp"
#include "xc setFamilyHistory.cpp"
#include "xc setImagingInfo.cpp"
#include "xc setReminder.cpp"
#include "xc setCholesterolReadings.cpp"
#include "xc setUnregisteredUser.cpp"
#include "xc_setUserPreference.cpp"

//Get Commands include
#include "xc getAddressInfo.cpp"
#include "xc getAdmit.cpp"
#include "xc getAllergyInfo.cpp"
#include "xc getBiographicsInfo.cpp"
#include "xc getCholesterolReadings.cpp"
#include "xc getCodes.cpp"
#include "xc getConvertPc.cpp"
#include "xc getCurrConvertPc.cpp"
#include "xc getCurrPreAdmit.cpp"
#include "xc getCurrLoa.cpp"
#include "xc getCurrTransfer.cpp"
#include "xc getDischarge.cpp"
#include "xc getEmploymentInfo.cpp"
#include "xc getCurrEncounter.cpp"
#include "xc getCurrEncounterid.cpp"
#include "xc getEncounterTree.cpp"
#include "xc getExternalIDs.cpp"
#include "xc_getFamilyTree.cpp"
#include "xc_getInsuranceInfo.cpp"
#include "xc_getLoa.cpp"
#include "xc_getMiscIDs.cpp"
#include "xc_getPhone.cpp"
#include "xc_getPhysicalInfo.cpp"
#include "xc_getPreAdmit.cpp"
#include "xc_getStats.cpp"
#include "xc_getTransfer.cpp"
#include "xc_getAccountInfo.cpp"
```

```
#include "xc getGuarantorInfo.cpp"
#include "xc getCareDirectives.cpp"
#include "xc getDisability.cpp"
#include "xc getInsuranceCoverage.cpp"
#include "xc getSecurityInfo.cpp"
#include "xc getDiagnosis.cpp"
#include "xc getPhysicianInfo.cpp"
#include "xc getPatientValuables.cpp"
#include "xc getLoaHistory.cpp"
#include "xc getDischargeHistory.cpp"
#include "xc getCpiIdExists.cpp"
#include "xc getCodeCats.cpp"
#include "xc getCpiId.cpp"
#include "xc getName.cpp"
#include "xc getPerson.cpp"
#include "xc getCompany.cpp"
#include "xc getNok.cpp"
#include "xc_getNokAll.cpp"
#include "xc_getNewEncounterId.cpp"
#include "xc_getPhysicians.cpp"
#include "xc_getFacilities.cpp"
#include "xc_getPocs.cpp"
#include "xc_getRooms.cpp"
#include "xc_getBeds.cpp"
#include "xc_getInsPlans.cpp"
#include "xc_getInsPlansByCompany.cpp"
#include "xc_getPatientStatus.cpp"
#include "xc_getPatientLocation.cpp"
#include "xc_getInPatients.cpp"
#include "xc_getPasswordReminder.cpp"
#include "xc_getIdealBPRanges.cpp"
#include "xc_getBloodPressureReadings.cpp"
#include "xc getPulseReadings.cpp"
#include "xc getWeightReadings.cpp"
#include "xc getSLMDLocations.cpp"
#include "xc getUserBiographics.cpp"
#include "xc getUserPhysicians.cpp"
#include "xc getUserInsurance.cpp"
#include "xc getHealthConditions.cpp"
#include "xc getImmunizations.cpp"
#include "xc getMedications.cpp"
#include "xc getSurgeryInfo.cpp"
#include "xc getTherapyInfo.cpp"
#include "xc getFamilyHistory.cpp"
#include "xc getImagingInfo.cpp"
#include "xc getReminder.cpp"
#include "xc getMassMailing.cpp"
#include "xc_getNewUnregUserId.cpp"
#include "xc getLifeclinicStats.cpp"
#include "xc_getUserPreference.cpp"

//Update Commands include
#include "xc uptAddressInfo.cpp"
#include "xc uptAdmit.cpp"
#include "xc uptCareDirectives.cpp"
#include "xc uptCompany.cpp"
#include "xc uptConvertPc.cpp"
#include "xc_uptDiagnosis.cpp"
#include "xc_uptDischarge.cpp"
#include "xc_uptEmployment.cpp"
#include "xc_uptEncounter.cpp"
#include "xc_uptEncounterHcp.cpp"
#include "xc_uptEncounterLog.cpp"
#include "xc_uptExternalCode.cpp"
#include "xc_uptFacility.cpp"
#include "xc_uptGuarantor.cpp"
#include "xc_uptInsurance.cpp"
```

```
#include "xc uptInsurancePlan.cpp"
#include "xc uptLoa.cpp"
#include "xc uptPatient.cpp"
#include "xc uptBiographics.cpp"
#include "xc uptPhone.cpp"
#include "xc uptPreAdmit.cpp"
#include "xc uptTransfer.cpp"
#include "xc uptIdMap.cpp"
#include "xc uptNok.cpp"
#include "xc uptMiscIds.cpp"
#include "xc uptName.cpp"
#include "xc uptPerson.cpp"
#include "xc_uptPhysical.cpp"

//Delete Commands Include
#include "xc delAudit.cpp"
#include "xc delDiagnosis.cpp"
#include "xc delHcpOffice.cpp"
#include "xc_delHcpSpecialty.cpp"
#include "xc_deletePhysical.cpp"
#include "xc_deleteEmploymentInfo.cpp"
#include "xc_deleteUserphysician.cpp"
#include "xc_deleteInsurance.cpp"
#include "xc_deleteAllergy.cpp"
#include "xc_deleteHealthConditions.cpp"
#include "xc_deleteImmunizations.cpp"
#include "xc_deleteMedications.cpp"
#include "xc_deleteSurgeryInfo.cpp"
#include "xc_deleteTherapyInfo.cpp"
#include "xc_deleteFamilyHistory.cpp"
#include "xc_deleteImagingInfo.cpp"
#include "xc_deleteReminder.cpp"
#include "xc_deleteUnregisteredUser.cpp"

//Insert Commands Include
#include "xc insCodeCategory.cpp"
#include "xc insCpiMaster.cpp"
#include "xc insCpiUser.cpp"
#include "xc insDiagnosis.cpp"
#include "xc insEncounterLog.cpp"
#include "xc insEncounterMap.cpp"
#include "xc insExternalCode.cpp"
#include "xc insInternalCode.cpp"
#include "xc insSysOrg.cpp"
#include "xc insEncounter.cpp"
#include "xc_insMassMailing.cpp"
```

```

#include "xc OtherCommands.h"
#include "Encryptor.h"
#include "rs_cpi_user.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_changePassword)

bool Cxc_changePassword::execCommand()
{
    bool fSuccess = true;
    CSdoConnection * pconn = NULL;

    try
    {
        Crs_cpi_user      rsCpiUser;

        string strCpiId;
        string strOldPassword, strNewPassword, strPassReminder;

        pconn = m_pcoClient->getConnection();
        pconn->beginTrans();

        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\"cpi_id\" is a required parameter.");
            throw fSuccess = false;
        }
        if (getParam("old_password", strOldPassword) == false)
        {
            m_emLast.setError("\"oldpassword\" is a required parameter");
            throw fSuccess = false;
        }
        if (getParam("new_password", strNewPassword) == false)
        {
            m_emLast.setError("\"newpassword\" is a required parameter");
            throw fSuccess = false;
        }
        if (getParam("password_reminder", strPassReminder) == false)
        {
            m_emLast.setError("\"password_reminder\" is a required parameter");
            throw fSuccess = false;
        }

        //check if cpi id is null.
        if (strCpiId.empty())
        {
            m_emLast.setError("\"cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        //get the user login,password.
        rsCpiUser.setActiveCommand("cmdFetchUserLogin");
        rsCpiUser.setParm("cpi_id", _variant_t (strCpiId.c_str()));

        if (pconn->execute(rsCpiUser) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }

        //check if user present.
        if (rsCpiUser.isEmpty())
        {
            m_emLast << "The user does not exist.";
            throw fSuccess = false;
        }
    }
}

```

```
}

//check if passed in old password is the valid one.
CEncryptor encryptor;
string strEncryptedPassword, strPassword;

strEncryptedPassword = (char *) ( bstr t) rsCpiUser.getField("password");
encryptor.Decrypt(strEncryptedPassword.c_str(), NULL, strPassword);

if (strOldPassword != strPassword)
{
    m_emLast.setError("Invalid Password.");
    throw fSuccess = false;
}

//encrypt new password
encryptor.Encrypt(strNewPassword.c_str(), NULL, strPassword);

//change password.
rsCpiUser.clearParms();
rsCpiUser.setRecordSetToNull();
rsCpiUser.setActiveCommand("cmdUpdate");
rsCpiUser.setParameter("cpi_id", _variant_t(strCpiId.c_str()));
rsCpiUser.setParameter("password", _variant_t(strPassword.c_str()));
rsCpiUser.setParameter("password_reminder", _variant_t(strPassReminder.c_str()));
rsCpiUser.setParameter("audit_id", _variant_t(getAuditId()));

if ((fSuccess = pconn->execute(rsCpiUser)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

}
catch(bool fError)
{
    fError;
}
catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}
catch(...)
{
    m_emLast.setError("Unkown exception raised. [Command:changePassword]");
    fSuccess = false;
}

//commit or Roll back the transaction.
if (pconn)
{
    if (fSuccess)    pconn->commitTrans();
    else             pconn->rollbackTrans();
}

return fSuccess;
}
```

```
#ifndef xc createuser h
#define xc_createuser_h

class Cxc_createUser : public CxcLCBroker
{
protected:
    enum { MASK VIP = 1,
           MASK ENCOUNTER = 2,
           MASK SYSTEM = 4,
           MASK HCP = 8,
           MASK_USER = 16
    };

public:
    virtual bool execCommand();
    CXC_DECLARE_FACTORY()
};

#endif
```

```
#include "xc DeleteCommands.h"
#include "rs_audit.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_delAudit)

bool Cxc_delAudit::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_audit    rsAudit;

        rsAudit.setActiveCommand("cmdDelete");

        //get connection
        CSdoConnection * pconn = m_pcoClient->getConnection();

        //execute
        if ((fSuccess = pconn->execute(rsAudit)) == false)
            m_emLast.setError(pconn->getLastError());

    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:delAudit]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc DeleteCommands.h"
#include "rs_diagnosis.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_delDiagnosis)

bool Cxc_delDiagnosis::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs diagnosis    rsDiagnosis;
        string strEncId, strRecId, strEncActionId;

        rsDiagnosis.setActiveCommand("cmdDelete");

        //required paramters.
        if (getParm("enc_id", strEncId) == false)
        {
            m_emLast.setError("\\"enc_id\\" is a required parameter.");
            return false;
        }
        if (getParm("rec_id", strRecId) == false)
        {
            m_emLast.setError("\\"rec_id\\" is a required parameter.");
            return false;
        }

        //set the parameters
        _variant_t vEncID(atol(strEncId.c_str()));
        _variant_t vRecID(atol(strRecId.c_str()));
        rsDiagnosis.setParameter("enc_id", variant t(vEncID));
        rsDiagnosis.setParameter("rec_id", _variant_t(vRecID));

        //set the optional parameter
        if (getParm("enc_action_id", strEncActionId) == true)
        {
            variant t vEncActionID(atol(strEncActionId.c_str()));
            rsDiagnosis.setParameter("enc_action_id", _variant_t(vEncActionID));
        }

        //get connection
        CSdoConnection * pconn = m_pcoClient->getConnection();

        //execute
        if ((fSuccess = pconn->execute(rsDiagnosis)) == false)
            m_emLast.setError(pconn->getLastError());

    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:delDiagnosis]");
        fSuccess = false;
    }

    return fSuccess;
}
```



```
#include "xc DeleteCommands.h"
#include "rs_allergy.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_deleteAllergy)

bool Cxc_deleteAllergy::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs allergy rs allergy;
        string strCpiId, strRecId;

        //get connection
        CSdoConnection * pconn = m_pcoClient->getConnection();

        //required paramater.
        if (getParm("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }

        //optional parameter
        getParm("rec_id", strRecId);

        long lCpiId = atol(strCpiId.c_str());
        long lRecId = atol(strRecId.c_str());

        if (!lCpiId)
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        //set the parameter
        if (lRecId)
        {
            rs allergy.setActiveCommand("cmdDeleteByRecId");
            rs_allergy.setParameter("rec_id", _variant_t(lRecId));
        }
        else
            rs_allergy.setActiveCommand("cmdDeleteAll");

        rs allergy.setParameter("cpi id", _variant_t(lCpiId));
        if (!pconn->execute(rs_allergy))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
    }

    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:deleteAllergy]");
        fSuccess = false;
    }
}
```

```
    }  
    return fSuccess;  
}
```

```
#ifndef xc_deleteCommands_h
#define xc_deleteCommands_h

#include "stdafx.h"
#include "xcLCBroker.h"

/////////////////////////////////////////////////////////////////
//
// Declaration of all the XML delete Commands Classes.
//
// Macro derives the class from CxcLCBroker
//
/////////////////////////////////////////////////////////////////

DECLARE_XML_DELETECMD_CLASS(Cxc_delAudit)
DECLARE_XML_DELETECMD_CLASS(Cxc_delDiagnosis)
DECLARE_XML_DELETECMD_CLASS(Cxc_delHcpOffice)
DECLARE_XML_DELETECMD_CLASS(Cxc_delHcpSpecialty)
DECLARE_XML_DELETECMD_CLASS(Cxc_deletePhysical)
DECLARE_XML_DELETECMD_CLASS(Cxc_deleteEmploymentInfo)
DECLARE_XML_DELETECMD_CLASS(Cxc_deleteUserPhysician)
DECLARE_XML_DELETECMD_CLASS(Cxc_deleteInsurance)
DECLARE_XML_DELETECMD_CLASS(Cxc_deleteAllergy)
DECLARE_XML_DELETECMD_CLASS(Cxc_deleteHealthConditions)
DECLARE_XML_DELETECMD_CLASS(Cxc_deleteImmunizations)
DECLARE_XML_DELETECMD_CLASS(Cxc_deleteMedications)
DECLARE_XML_DELETECMD_CLASS(Cxc_deleteSurgeryInfo)
DECLARE_XML_DELETECMD_CLASS(Cxc_deleteTherapyInfo)
DECLARE_XML_DELETECMD_CLASS(Cxc_deleteFamilyHistory)
DECLARE_XML_DELETECMD_CLASS(Cxc_deleteImagingInfo)
DECLARE_XML_DELETECMD_CLASS(Cxc_deleteReminder)
DECLARE_XML_DELETECMD_CLASS(Cxc_deleteUnregisteredUser)

#endif
```

```
#include "xc DeleteCommands.h"
#include "rs employers.h"
#include "rs_cpi_master.h"
```

```
////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_deleteEmploymentInfo)
```

```
bool Cxc_deleteEmploymentInfo::execCommand()
{
    bool fSuccess = true;
    CSdoConnection * pconn = NULL;

    try
    {
        Crs employers rsEmployers;
        Crs cpi master rsCpiMaster;
        string strCpiId, strRecId, strEmployerId;

        //required parameters
        if (getParm("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\"cpi_id\" is a required parameter.");
            throw fSuccess = false;
        }
        if (getParm("rec_id", strRecId) == false)
        {
            m_emLast.setError("\"rec_id\" is a required parameter.");
            throw fSuccess = false;
        }

        //optional parameter
        getParm("employer_id", strEmployerId);

        if (strCpiId.empty())
        {
            m_emLast.setError("\"cpi_id\" is NULL.");
            throw fSuccess = false;
        }
        if (strRecId.empty())
        {
            m_emLast.setError("\"rec_id\" is NULL.");
            throw fSuccess = false;
        }

        //get connection
        pconn = m_pcoClient->getConnection();

        //start transaction
        pconn->beginTrans();

        //delete the employment record
        rsEmployers.clearParms();
        rsEmployers.setRecordSetToNull();
        rsEmployers.setActiveCommand("cmdDelete");
        rsEmployers.setParameter("cpi_id", variant t(atol(strCpiId.c_str())));
        rsEmployers.setParameter("rec_id", _variant_t(atol(strRecId.c_str())));
        if (!pconn->execute(rsEmployers))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }

        //if employer_id is provided, delete employer information.
    }
}
```

```
long lEmployerId = atol(strEmployerId.c_str());
if (lEmployerId)
{
    //check if the employer is referenced by other users

    rsEmployers.setActiveCommand("cmdCheckReferenceExist");
    rsEmployers.setParameter("employer_id",_variant_t(lEmployerId));
    if (!pconn->execute(rsEmployers))
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }

    //if employer is not referenced then delete the employer
    if (rsEmployers.isEmpty())
    {
        rsCpiMaster.setActiveCommand("cmdDelete");
        rsCpiMaster.setParameter("cpi_id",_variant_t(lEmployerId));
        if (!pconn->execute(rsCpiMaster))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
    }
    else
        rsEmployers.setRecordSetToNull();
}

}
catch(bool fError)
{
    fError;
}
catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}
catch(...)
{
    m_emLast.setError("Unkown exception raised. [Command:deleteEmploymentInfo]");
    fSuccess = false;
}

//commit or Roll back the transaction.
if (pconn)
{
    if (fSuccess)    pconn->commitTrans();
    else            pconn->rollbackTrans();
}

return fSuccess;
}
```

```
#include "xc DeleteCommands.h"
#include "rs_family_history.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_deleteFamilyHistory)

bool Cxc_deleteFamilyHistory::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs family history rs family_history;
        string strCpiId, strRecId;

        //get connection
        CSdoConnection * pconn = m_pcoClient->getConnection();

        //required paramater.
        if (getParm("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }

        //optional parameter
        getParm("rec_id", strRecId);

        long lCpiId = atol(strCpiId.c_str());
        long lRecId = atol(strRecId.c_str());

        if (!lCpiId)
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        //set the parameter
        if (lRecId)
        {
            rs family_history.setActiveCommand("cmdDeleteByRecId");
            rs_family_history.setParameter("rec_id", _variant_t(lRecId));
        }
        else
            rs_family_history.setActiveCommand("cmdDeleteAll");

        rs family_history.setParameter("cpi id", _variant_t(lCpiId));
        if (!pconn->execute(rs_family_history))
        {
            m_emLast.setError(pconn->getLastErrorMessage());
            throw fSuccess = false;
        }
    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:deleteFamilyHistory]");
        fSuccess = false;
    }
}
```

```
    }  
    return fSuccess;  
}
```

```
#include "xc DeleteCommands.h"
#include "rs_health_condition.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_deleteHealthConditions)

bool Cxc_deleteHealthConditions::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs health condition    rs_hc;
        string strCpiId, strRecId;

        //get connection
        CSdoConnection * pconn = m_pcoClient->getConnection();

        //required paramater.
        if (getParm("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }

        //optional parameter
        getParm("rec_id", strRecId);

        long lCpiId = atol(strCpiId.c_str());
        long lRecId = atol(strRecId.c_str());

        if (!lCpiId)
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        //set the parameter
        if (lRecId)
        {
            rs_hc.setActiveCommand("cmdDeleteByRecId");
            rs_hc.setParameter("rec_id", _variant_t(lRecId));
        }
        else
            rs_hc.setActiveCommand("cmdDeleteAll");

        rs_hc.setParameter("cpi id", _variant_t(lCpiId));
        if (!pconn->execute(rs_hc))
        {
            m_emLast.setError(pconn->getLastErrorMessage());
            throw fSuccess = false;
        }
    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:deleteHealthConditions]");
        fSuccess = false;
    }
}
```



```
    }  
    return fSuccess;  
}
```

```
#include "xc_DeleteCommands.h"
#include "rs_imaging.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_deleteImagingInfo)

bool Cxc_deleteImagingInfo::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs imaging rs imaging;
        string strCpiId, strRecId;

        //get connection
        CSdoConnection * pconn = m_pcoClient->getConnection();

        //required paramater.
        if (getParm("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }

        //optional parameter
        getParm("rec_id", strRecId);

        long lCpiId = atol(strCpiId.c_str());
        long lRecId = atol(strRecId.c_str());

        if (!lCpiId)
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        //set the parameter
        if (lRecId)
        {
            rs imaging.setActiveCommand("cmdDeleteByRecId");
            rs_imaging.setParameter("rec_id", _variant_t(lRecId));
        }
        else
            rs_imaging.setActiveCommand("cmdDeleteAll");

        rs imaging.setParameter("cpi_id", _variant_t(lCpiId));
        if (!pconn->execute(rs_imaging))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:deleteImagingInfo]");
        fSuccess = false;
    }
}
```

```
    }  
    return fSuccess;  
}
```

```
#include "xc_DeleteCommands.h"
#include "rs_immunization.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_deleteImmunizations)

bool Cxc_deleteImmunizations::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs immunization    rs immunization;
        string strCpiId, strRecId;

        //get connection
        CSdoConnection * pconn = m_pcoClient->getConnection();

        //required paramater.
        if (getParm("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }

        //optional parameter
        getParm("rec_id", strRecId);

        long lCpiId = atol(strCpiId.c_str());
        long lRecId = atol(strRecId.c_str());

        if (!lCpiId)
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        //set the parameter
        if (lRecId)
        {
            rs immunization.setActiveCommand("cmdDeleteByRecId");
            rs_immunization.setParameter("rec_id", _variant_t(lRecId));
        }
        else
            rs_immunization.setActiveCommand("cmdDeleteAll");

        rs immunization.setParameter("cpi id", _variant_t(lCpiId));
        if (!pconn->execute(rs_immunization))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:deleteImmunizations]");
        fSuccess = false;
    }
}
```

```
    }  
    return fSuccess;  
}
```

```

#include "xc_DeleteCommands.h"

#include "rs_insurance.h"
#include "rs_cpi_master.h"
#include "rs_employers.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_deleteInsurance)

bool Cxc_deleteInsurance::execCommand()
{
    bool fSuccess = true;
    CSdoConnection * pconn = NULL;

    try
    {
        Crs_cpi_master    rs_cpi_master;
        Crs_employers     rs_employers;
        Crs_insured       rs_insured;
        Crs_insurance     rs_insurance;
        Crs_insured_dependents rs_insured_dependents;

        string strCpiId, strRecId, strSubId, strCompanyId;

        //required parameters
        if (getParm("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            throw fSuccess = false;
        }
        if (getParm("rec_id", strRecId) == false)
        {
            m_emLast.setError("\rec_id\" is a required parameter.");
            throw fSuccess = false;
        }

        //optional parameter
        getParm("subscriber_id", strSubId);
        getParm("company_id", strCompanyId);

        if (strCpiId.empty())
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }
        if (strRecId.empty())
        {
            m_emLast.setError("\rec_id\" is NULL.");
            throw fSuccess = false;
        }

        long lCpiId = atol(strCpiId.c_str());
        long lSubCpiId = atol(strSubId.c_str());
        long lRecId = atol(strRecId.c_str());
        long lCompanyId = atol(strCompanyId.c_str());

        //get connection
        pconn = m_pcoClient->getConnection();

        //start transaction
        pconn->beginTrans();

        //////////////////////////////////
    }
}

```

```

//check if self insured or dependent and delete insurance records
////////////////////////////////////

//if subscriber_id provided, user is dependent.
if (lSubCpiId)
{
    //check if subscriber is referenced by other users.
    rs_insured_dependents.clearParms();
    rs_insured_dependents.setRecordSetToNull();
    rs_insured_dependents.setActiveCommand("cmdCheckReferenceExist");
    rs_insured_dependents.setParameter("insured_cpi_id", _variant_t(lSubCpiId));
    rs_insured_dependents.setParameter("dependent_cpi_id", _variant_t(lCpiId));
    if (!pconn->execute(rs_insured_dependents))
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }

    //delete subscriber from database if not referenced
    if (rs_insured_dependents.isEmpty())
    {
        rs_cpi_master.setActiveCommand("cmdDelete");
        rs_cpi_master.setParameter("cpi_id", _variant_t(lSubCpiId));
        if (!pconn->execute(rs_cpi_master))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
    }
    else
    {
        //delete only the insured_dependent record of subscriber for this user.
        rs_insured_dependents.clearParms();
        rs_insured_dependents.setRecordSetToNull();
        rs_insured_dependents.setActiveCommand("cmdDelete");
        rs_insured_dependents.setParameter("insured_cpi_id", _variant_t
(lSubCpiId));
        rs_insured_dependents.setParameter("dependent_cpi_id", _variant_t
(lCpiId));
        if (!pconn->execute(rs_insured_dependents))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
    }
}
else
{
    //user is self insured. delete user record from insured table.
    rs_insured.clearParms();
    rs_insured.setRecordSetToNull();
    rs_insured.setActiveCommand("cmdDelete");
    rs_insured.setParameter("cpi_id", _variant_t (lCpiId));
    rs_insured.setParameter("rec_id", _variant_t (lRecId));
    if (!pconn->execute(rs_insured))
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }
    rs_insured.setRecordSetToNull();
}

//delete company if id provided.
if (lCompanyId)
{

```

```

        bool fReferenced = false;

        //check if company is referenced ,
        //(i.e. check if the plan id's of this company are referenced)
        rs_insurance.clearParms();
        rs_insurance.setRecordSetToNull();
        rs_insurance.setActiveCommand("cmdCheckCompanyRefExist");
        rs_insurance.setParameter("company_id",_variant_t(lCompanyId));
        if (!pconn->execute(rs_insurance))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }

        if (!rs_insurance.isEmpty())
        {
            fReferenced = true;
            rs_insurance.setRecordSetToNull();
        }

        //also check if company is referenced as an employer
        rs_employers.clearParms();
        rs_employers.setRecordSetToNull();
        rs_employers.setActiveCommand("cmdCheckReferenceExist");
        rs_employers.setParameter("employer_id",_variant_t(lCompanyId));
        if (!pconn->execute(rs_employers))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }

        if (!rs_employers.isEmpty())
        {
            fReferenced = true;
            rs_employers.setRecordSetToNull();
        }

        //delete company from database if not referenced
        if (!fReferenced)
        {
            rs_cpi_master.setActiveCommand("cmdDelete");
            rs_cpi_master.setParameter("cpi_id",_variant_t(lCompanyId));
            if (!pconn->execute(rs_cpi_master))
            {
                m_emLast.setError(pconn->getLastError());
                throw fSuccess = false;
            }
        }
    }

}

catch(bool fError)
{
    fError;
}

catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}

catch(...)
{
    m_emLast.setError("Unkown exception raised. [Command:deleteInsurance]");
    fSuccess = false;
}

```



```
//commit or Roll back the transaction.  
if (pconn)  
{  
    if (fSuccess)    pconn->commitTrans();  
    else             pconn->rollbackTrans();  
}  
  
return fSuccess;  
}
```

```
#include "xc_DeleteCommands.h"
#include "rs_medication.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_deleteMedications)

bool Cxc_deleteMedications::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_medication rs_medication;
        string strCpiId, strRecId;

        //get connection
        CSdoConnection * pconn = m_pcoClient->getConnection();

        //required paramater.
        if (getParm("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\"cpi_id\" is a required parameter.");
            return false;
        }

        //optional parameter
        getParm("rec_id", strRecId);

        long lCpiId = atol(strCpiId.c_str());
        long lRecId = atol(strRecId.c_str());

        if (!lCpiId)
        {
            m_emLast.setError("\"cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        //set the parameter
        if (lRecId)
        {
            rs_medication.setActiveCommand("cmdDeleteByRecId");
            rs_medication.setParameter("rec_id", _variant_t(lRecId));
        }
        else
            rs_medication.setActiveCommand("cmdDeleteAll");

        rs_medication.setParameter("cpi_id", _variant_t(lCpiId));
        if (!pconn->execute(rs_medication))
        {
            m_emLast.setError(pconn->getLastErrorMessage());
            throw fSuccess = false;
        }
    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:deleteMedications]");
        fSuccess = false;
    }
}
```

```
    }  
    return fSuccess;  
}
```

```
#include "xc_DeleteCommands.h"
#include "rs_physical.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_deletePhysical)

bool Cxc_deletePhysical::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_physical    rsPhysical;
        string strCpiId, strRecId;

        //get connection
        CSdoConnection * pconn = m_pcoClient->getConnection();

        //required paramater.
        if (getParm("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }

        if (strCpiId.empty())
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            return false;
        }

        //set the parameter
        rsPhysical.setActiveCommand("cmdDelete");
        rsPhysical.setParameter("cpi_id", _variant_t(atol(strCpiId.c_str())));

        //execute
        if ((fSuccess = pconn->execute(rsPhysical)) == false)
            m_emLast.setError(pconn->getLastError());

    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:deletePhysical]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_DeleteCommands.h"
#include "rs_reminders.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_deleteReminder)

bool Cxc_deleteReminder::execCommand()
{
    bool fSuccess = false;
    bool fTransInProgress = false;
    CSdoConnection * pconn;

    try
    {
        Crs_reminder    rs_reminder;
        string strCpiId, strRecId;

        //get connection
        pconn = m_pcoClient->getConnection();

        //required paramater.
        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }

        //optional parameter
        getParam("rec_id", strRecId);

        long lCpiId = atol(strCpiId.c_str());
        long lRecId = atol(strRecId.c_str());

        if (!lCpiId)
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        //set the parameter
        if (lRecId)
        {
            rs_reminder.setActiveCommand("cmdDeleteByRecId");
            rs_reminder.setParameter("cpi_id", _variant_t(lCpiId));
            rs_reminder.setParameter("rec_id", _variant_t(lRecId));
        }
        else
        {
            rs_reminder.setActiveCommand("cmdDeleteByCpiId");
            rs_reminder.setParameter("rm_cpi_id", _variant_t(lCpiId));
            rs_reminder.setParameter("me_cpi_id", _variant_t(lCpiId));
        }

        pconn->beginTrans();

        fTransInProgress = true;

        if (!pconn->execute(rs_reminder))
        {
            m_emLast.setError(pconn->getLastErrorMessage());
            throw fSuccess = false;
        }

        fSuccess = true;
    }
    catch(bool fError)
    {
    }
}
```

```
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:deleteReminder]");
        fSuccess = false;
    }

    if (fTransInProgress)
    {
        if (fSuccess)
            pconn->commitTrans();
        else
            pconn->rollbackTrans();
    }

    return fSuccess;
}
```

```
#include "xc_DeleteCommands.h"
#include "rs_surgery.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_deleteSurgeryInfo)

bool Cxc_deleteSurgeryInfo::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_surgery rs_surgery;
        string strCpiId, strRecId;

        //get connection
        CSdoConnection * pconn = m_pcoClient->getConnection();

        //required paramater.
        if (getParm("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }

        //optional parameter
        getParm("rec_id", strRecId);

        long lCpiId = atol(strCpiId.c_str());
        long lRecId = atol(strRecId.c_str());

        if (!lCpiId)
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        //set the parameter
        if (lRecId)
        {
            rs_surgery.setActiveCommand("cmdDeleteByRecId");
            rs_surgery.setParameter("rec_id", _variant_t(lRecId));
        }
        else
            rs_surgery.setActiveCommand("cmdDeleteAll");

        rs_surgery.setParameter("cpi_id", _variant_t(lCpiId));
        if (!pconn->execute(rs_surgery))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:deleteSurgeryInfo]");
        fSuccess = false;
    }
}
```

```
    }  
    return fSuccess;  
}
```



```
#include "xc_DeleteCommands.h"
#include "rs_therapy.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_deleteTherapyInfo)

bool Cxc_deleteTherapyInfo::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_therapy rs_therapy;
        string strCpiId, strRecId;

        //get connection
        CSdoConnection * pconn = m_pcoClient->getConnection();

        //required paramater.
        if (getParm("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }

        //optional parameter
        getParm("rec_id", strRecId);

        long lCpiId = atol(strCpiId.c_str());
        long lRecId = atol(strRecId.c_str());

        if (!lCpiId)
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        //set the parameter
        if (lRecId)
        {
            rs_therapy.setActiveCommand("cmdDeleteByRecId");
            rs_therapy.setParameter("rec_id", _variant_t(lRecId));
        }
        else
            rs_therapy.setActiveCommand("cmdDeleteAll");

        rs_therapy.setParameter("cpi_id", _variant_t(lCpiId));
        if (!pconn->execute(rs_therapy))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:deleteTherapyInfo]");
        fSuccess = false;
    }
}
```

```
    }  
    return fSuccess;  
}
```

```
#include "xc_DeleteCommands.h"

#include "rs_unregistered_user.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_deleteUnregisteredUser)

bool Cxc_deleteUnregisteredUser::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_unregistered_user rsUnregUser;
        string strUserId;

        //get the user_id.
        if (getParam("user_id", strUserId) == false)
        {
            m_emLast.setError("\\"user_id\\" is a required parameter.");
            throw fSuccess = false;
        }

        //get db connection
        CSdoConnection * pconn = m_pcoClient->getConnection();

        //////////////////////////////////////
        // Delete the unregistered user record.
        //////////////////////////////////////

        rsUnregUser.clearParms();
        rsUnregUser.setRecordSetToNull();
        rsUnregUser.setActiveCommand("cmdDelete");
        rsUnregUser.setParameter("user_id", _variant_t(atol(strUserId.c_str())));
        if ((fSuccess = pconn->execute(rsUnregUser)) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:deleteUnregisteredUser]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_DeleteCommands.h"

#include "rs_hcp.h"
#include "rs_cpi_master.h"
#include "rs_encounter.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_deleteUserPhysician)

bool Cxc_deleteUserPhysician::execCommand()
{
    bool fSuccess = true;
    CSdoConnection * pconn = NULL;

    try
    {
        Crs_encounter_hcp    rs_encounter_hcp;
        Crs_cpi_master        rs_cpi_master;
        Crs_encounter         rs_encounter;
        string strCpiId, strRecId, strHcpId;

        //required parameters
        if (getParm("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\ is a required parameter.");
            throw fSuccess = false;
        }
        if (getParm("rec_id", strRecId) == false)
        {
            m_emLast.setError("\rec_id\ is a required parameter.");
            throw fSuccess = false;
        }

        //optional parameter
        getParm("physician_id", strHcpId);

        if (strCpiId.empty())
        {
            m_emLast.setError("\cpi_id\ is NULL.");
            throw fSuccess = false;
        }
        if (strRecId.empty())
        {
            m_emLast.setError("\rec_id\ is NULL.");
            throw fSuccess = false;
        }

        long lEncId, lRecId;

        lEncId = 0;
        lRecId = atol(strRecId.c_str());

        //get connection
        pconn = m_pcoClient->getConnection();

        //start transaction
        pconn->beginTrans();

        //get the default encounter id for the user. [Assumed that user will have only 1
encouter]
        rs_encounter.clearParms();
        rs_encounter.setRecordSetToNull();
        rs_encounter.setActiveCommand("cmdFetchCurrentId");
        rs_encounter.setParameter("cpi_id", _variant_t(atol(strCpiId.c_str())));
        if (!pconn->execute(rs_encounter))
        {
            m_emLast.setError("Failed to execute command.");
            throw fSuccess = false;
        }
    }
}
```

```
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }

    //get encounter id
    if(!rs_encounter.isEmpty())
    {
        string strEncId;
        rs_encounter.getField("enc_id", strEncId);
        lEncId = atol(strEncId.c_str());
        rs_encounter.setRecordSetToNull();
    }

    //check if default encounter present.
    if (!lEncId)
    {
        m_emLast.setError("No physician records present.");
        throw fSuccess = false;
    }

    //delete the physician record
    rs_encounter_hcp.clearParms();
    rs_encounter_hcp.setRecordSetToNull();
    rs_encounter_hcp.setActiveCommand("cmdDelete");
    rs_encounter_hcp.setParameter("enc_id", _variant_t(lEncId));
    rs_encounter_hcp.setParameter("rec_id", _variant_t(lRecId));
    if (!pconn->execute(rs_encounter_hcp))
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }

    //if physician_id is provided, delete physician information.
    long lHcpId = atol(strHcpId.c_str());
    if (lHcpId)
    {
        //check if the physician is referenced by other users

        rs_encounter_hcp.setActiveCommand("cmdCheckReferenceExist");
        rs_encounter_hcp.setParameter("hcp_id", _variant_t(lHcpId));
        if (!pconn->execute(rs_encounter_hcp))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }

        //if physician is not referenced then delete the physician
        if (rs_encounter_hcp.isEmpty())
        {
            rs_cpi_master.setActiveCommand("cmdDelete");
            rs_cpi_master.setParameter("cpi_id", _variant_t(lHcpId));
            if (!pconn->execute(rs_cpi_master))
            {
                m_emLast.setError(pconn->getLastError());
                throw fSuccess = false;
            }
        }
        else
            rs_encounter_hcp.setRecordSetToNull();
    }
}

catch(bool fError)
```

```
{
    fError;
}
catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}
catch(...)
{
    m_emLast.setError("Unkown exception raised. [Command:deleteUserPhysician]");
    fSuccess = false;
}

//commit or Roll back the transaction.
if (pconn)
{
    if (fSuccess)    pconn->commitTrans();
    else            pconn->rollbackTrans();
}

return fSuccess;
}
```

```
#include "xc_DeleteCommands.h"
#include "rs_hcp.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_delHcpOffice)

bool Cxc_delHcpOffice::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_hcp_office rsHcpOffice;
        string strCpiId, strRecId;

        rsHcpOffice.setActiveCommand("cmdDelete");

        //required parameter.
        if (getParm("cpi_Id", strCpiId) == false)
        {
            m_emLast.setError("\\"cpi_id\\" is a required parameter.");
            return false;
        }

        //set the parameter
        _variant_t vCpiID(atol(strCpiId.c_str()));
        rsHcpOffice.setParameter("cpi_id", _variant_t(vCpiID));

        //set the optional parameter
        if (getParm("rec_id", strRecId) == true)
        {
            _variant_t vRecID(atol(strRecId.c_str()));
            rsHcpOffice.setParameter("rec_id", _variant_t(vRecID));
        }

        //get connection
        CSdoConnection * pconn = m_pcoClient->getConnection();

        //execute
        if ((fSuccess = pconn->execute(rsHcpOffice)) == false)
            m_emLast.setError(pconn->getLastErrorMessage());

    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:delHcpOffice]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_DeleteCommands.h"
#include "rs_hcp.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_delHcpSpecialty)

bool Cxc_delHcpSpecialty::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_hcp_specialty rsHcp;
        string strCpiId, strSpecialtyId;

        rsHcp.setActiveCommand("cmdDelete");

        //required parameter.
        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\\"cpi_id\\" is a required parameter.");
            return false;
        }

        //set the parameter
        _variant_t vCpiID(atol(strCpiId.c_str()));
        rsHcp.setParameter("cpi_id", _variant_t(vCpiID));

        //set the optional parameter
        if (getParam("specialty_id", strSpecialtyId) == true)
        {
            _variant_t vSpecialtyID(atol(strSpecialtyId.c_str()));
            rsHcp.setParameter("specialty_id", _variant_t(vSpecialtyID));
        }

        //get connection
        CSdoConnection * pconn = m_pccClient->getConnection();

        //execute
        if ((fSuccess = pconn->execute(rsHcp)) == false)
            m_emLast.setError(pconn->getLastError());

    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:delHcpSpeciality]");
        fSuccess = false;
    }

    return fSuccess;
}
```



```
#ifndef _xc_execSearch_h
#define _xc_execSearch_h

class Cxc_execSearch : protected CxcLCBroker
{
public:
    virtual bool parseCommand(CXmlDocument * pdoc);
    virtual bool execCommand();
    CXC_DECLARE_FACTORY()
};

#endif
```

```
#include "xc_getCommands.h"
#include "rs_account.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getAccountInfo)

bool Cxc_getAccountInfo::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_account rsAccount;
        string strEncID;

        rsAccount.setActiveCommand("cmdFetch");

        if (getParam("enc_id", strEncID) == false)
        {
            m_emLast.setError("\\"enc_id\\" is a required parameter.");
            return false;
        }
        _variant_t vEncID(atol(strEncID.c_str()));
        rsAccount.setParameter("enc_id", vEncID);

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsAccount)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getAccountInfo/>");
            fSuccess = rsAccount.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getAccountInfo]");
        fSuccess = false;
    }
}

return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_address.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getAddressInfo)

bool Cxc_getAddressInfo::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_address rsAddress;
        string      strCpiID;
        string      strActiveSw;

        rsAddress.setActiveCommand("cmdFetch");

        if (getParm("cpi_id", strCpiID) == false)
        {
            m_emLast.setError("\\"cpi_id\\" is a required parameter.");
            return false;
        }
        _variant_t vCpiID(atol(strCpiID.c_str()));
        rsAddress.setParameter("cpi_id", vCpiID);

        if (getParm("active_sw", strActiveSw) == true)
        {
            _variant_t vActiveSw(atol(strActiveSw.c_str()));
            rsAddress.setParameter("active_sw", vActiveSw);
        }

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsAddress)) == false)
            m_emLast.setError(pconn->getLastErrorMessage());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getAddressInfo/>");
            fSuccess = rsAddress.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getAddressInfo]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_admit.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getAdmit)

bool Cxc_getAdmit::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_admit    rsAdmit;
        string strEncId;

        rsAdmit.setActiveCommand("cmdFetch");

        if (getParm("enc_id", strEncId) == false)
        {
            m_emLast.setError("\\"enc_id\\" is a required parameter.");
            return false;
        }

        long lEncId = atol(strEncId.c_str());
        rsAdmit.setParameter("enc_id", _variant_t(lEncId));

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsAdmit)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getAdmit/>");
            fSuccess = rsAdmit.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getAdmitInfo]");
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_allergy.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getAllergyInfo)

bool Cxc_getAllergyInfo::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_allergy      rsAllergies;
        string strCpiID;

        if (getParam("cpi_id", strCpiID) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }

        CSdoConnection * pconn = m_pcoClient->getConnection();

        //get allergies
        rsAllergies.setActiveCommand("cmdFetch");
        rsAllergies.setParameter("cpi_id", _variant_t(atol(strCpiID.c_str())));
        if ((fSuccess = pconn->execute(rsAllergies)) == false)
        {
            m_emLast.setError(pconn->getLastError());
        }
        else
        {
            m_pdocResults = new CXmlDocument("<getAllergyInfo/>");
            fSuccess = rsAllergies.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getAllergyInfo]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_location.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getBeds)

bool Cxc_getBeds::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_bed rs_bed;

        rs_bed.setActiveCommand("cmdFetchAll");

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rs_bed)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getBeds/>");
            fSuccess = rs_bed.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getBeds]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_name.h"
#include "rs_person.h"
```

```
////////////////////////////////////
```

```
CXC_IMPLEMENT_FACTORY(Cxc_getBiographicalInfo)
```

```
bool Cxc_getBiographicalInfo::execCommand()
```

```
{
    bool fSuccess = false;

    try
    {
        Crs_name      rsName;
        Crs_person     rsPerson;

        string strCpiID;
        string strActiveSw;

        if (getParm("cpi_id", strCpiID) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }
        _variant_t vCpiID(atol(strCpiID.c_str()));
        rsName.setActiveCommand("cmdFetch");
        rsName.setParameter("cpi_id", vCpiID);

        if (getParm("active_sw", strActiveSw) == true)
        {
            _variant_t vActiveSw(atol(strActiveSw.c_str()));
            rsName.setParameter("active_sw", vActiveSw);
        }

        rsPerson.setActiveCommand("cmdFetch");
        rsPerson.setParameter("cpi_id", vCpiID);

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsName)) == false)
        {
            m_emLast.setError(pconn->getLastError());
        }
        else
        {
            if ((fSuccess = pconn->execute(rsPerson)) == false)
            {
                m_emLast.setError(pconn->getLastError());
            }
        }

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getBiographicsInfo/>");
            fSuccess = rsName.toXml(*m_pdocResults);
            fSuccess = rsPerson.toXml(*m_pdocResults);
        }
    }
}
```

```
/*
////////////////////////////////////
//XML read testing code
```

```
    CXmlElement elroot, eltable, eltag, elrow, eldata;
    string strTag, strData;
```

```
bool fFound = false;

m_pdocResults->getRoot(&elroot);

//get name tag
fFound = elroot.getFirst(&eltable);
eltable.getTag(strTag);

//get row tag
eltable.getFirst(&elrow);
elrow.getTag(strTag);

//get data
fFound = elrow.getFirstItem("cpi_id", &eldata);
if (fFound) eldata.getText(strData);
fFound = elrow.getFirstItem("last_name", &eldata);
if (fFound) eldata.getText(strData);
fFound = elrow.getFirstItem("first_name", &eldata);
if (fFound) eldata.getText(strData);
fFound = elrow.getFirstItem("middle_name", &eldata);
if (fFound) eldata.getText(strData);
fFound = elrow.getFirstItem("nick_name", &eldata);
if (fFound) eldata.getText(strData);

////////////////////////////////////

*/
}
catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}
catch(...)
{
    m_emLast.setError("Unkown exception raised. [Command:getBiographicalInfo]");
    fSuccess = false;
}

return fSuccess;
}
```



```
#include "xc_GetCommands.h"
#include "rs_blood_pressure.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getBloodPressureReadings)

bool Cxc_getBloodPressureReadings::execCommand()
{
    bool fSuccess = true;

    try
    {
        Crs_blood_pressure rsBP;
        string strCpiId, strStartDate, strEndDate;

        CSdoConnection * pconn = m_pcoClient->getConnection();

        //get cpi_id
        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            throw fSuccess = false;
        }

        //optional parameters.
        getParam("start_dt", strStartDate);
        getParam("end_dt", strEndDate);

        //check if cpi_id is null.
        if (strCpiId.empty())
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }
        long lCpiId = atol(strCpiId.c_str());

        //if start_dt is not provided, then fetch all bp records.
        if (strStartDate.empty())
        {
            //fetch all bp records.
            rsBP.setActiveCommand("cmdFetchBPAll");
            rsBP.setParam("cpi_id", _variant_t (lCpiId));
        }
        else
        {
            COleDateTime oledate;
            DATE dtStartDate;

            //parse start_dt.
            oledate.ParseDateTime(strStartDate.c_str());
            dtStartDate = (DATE) oledate;
            if (dtStartDate == NULL)
            {
                m_emLast.setError("\start_dt\" is Invalid.");
                throw fSuccess = false;
            }

            if (strEndDate.empty())
            {
                //fetch all bp records.
                rsBP.setActiveCommand("cmdFetchBPByStartDate");
                rsBP.setParam("cpi_id", _variant_t (lCpiId));
                rsBP.setParam("start_dt", _variant_t (dtStartDate));
            }
        }
    }
}
```

```
else
{
    DATE dtEndDate;

    //parse end_dt.
    oledate.ParseDateTime(strEndDate.c_str());
    dtEndDate = (DATE) oledate;
    if (dtEndDate == NULL)
    {
        m_emLast.setError("\end_dt\" is Invalid.");
        throw fSuccess = false;
    }

    //get records from start_dt to end_dt
    rsBP.SetActiveCommand("cmdFetchBPByDateRange");
    rsBP.setParm("cpi_id", _variant_t (lCpiId));
    rsBP.setParm("start_dt", _variant_t (dtStartDate));
    rsBP.setParm("end_dt", _variant_t (dtEndDate));
}

if (pconn->execute(rsBP) == false)
{
    m_emLast.setError(pconn->getLastErrorMessage());
    throw fSuccess = false;
}
else
{
    m_pdocResults = new CXmlDocument("<getBloodPressureReadings/>");
    fSuccess = rsBP.toXml(*m_pdocResults);
}

}

catch(bool fError)
{
    fError;
}

catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}

catch(...)
{
    m_emLast.setError("Unkown exception raised. [Command:getBloodPressureReadings]");
    fSuccess = false;
}

return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_care_directives.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getCareDirectives)

bool Cxc_getCareDirectives::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_care_directives rsCareDirectives;
        string strCpiID;

        rsCareDirectives.setActiveCommand("cmdFetch");

        if (getParam("cpi_id", strCpiID) == false)
        {
            m_emLast.setError("\\"cpi_id\\" is a required parameter.");
            return false;
        }

        _variant_t vCpiID(atol(strCpiID.c_str()));
        rsCareDirectives.setParameter("cpi_id", vCpiID);

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsCareDirectives)) == false)
            m_emLast.setError(pconn->getLastErrorMessage());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getCareDirectives/>");
            fSuccess = rsCareDirectives.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getCareDirectives]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_GetCommands.h"
#include "rs_cholesterol.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getCholesterolReadings)

bool Cxc_getCholesterolReadings::execCommand()
{
    bool fSuccess = true;

    try
    {
        Crs_cholesterol rsCholesterol;

        string strCpiId;
        string strStartDate;

        CSdoConnection * pconn = m_pcoClient->getConnection();

        //get cpi_id
        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            throw fSuccess = false;
        }

        //check if cpi_id is null.
        if (strCpiId.empty())
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }
        long lCpiId = atol(strCpiId.c_str());

        //optional parameters.
        getParam("start_dt", strStartDate);

        //if start_dt is not provided, then fetch all bp records.
        if (strStartDate.empty())
        {
            //fetch all bp records.
            rsCholesterol.setActiveCommand("cmdFetchCholAll");
            rsCholesterol.setParm("cpi_id", _variant_t (lCpiId));
        }
        else
        {
            //get records from start_dt to end_dt

            COleDateTime odtStart;
            DATE dateStart;

            //parse start_dt.
            odtStart.ParseDateTime(strStartDate.c_str());
            dateStart = (DATE) odtStart;
            if (dateStart == NULL)
            {
                m_emLast.setError("\start_dt\" is Invalid.");
                throw fSuccess = false;
            }

            //fetch all bp records.
            rsCholesterol.setActiveCommand("cmdFetchCholByDateRange");
            rsCholesterol.setParm("cpi_id", _variant_t (lCpiId));
            rsCholesterol.setParm("start_dt", _variant_t (dateStart));
        }
    }
}
```

```
        if (pconn->execute(rsCholesterol) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
        else
        {
            m_pdocResults = new CXmlDocument("<getCholesterolReadings/>");
            fSuccess = rsCholesterol.toXml(*m_pdocResults);
        }

    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getBloodPressureReadings]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#ifndef xc_getCodeCats_h
#define xc_getCodeCats_h

class Cxc_getCodeCats : public CxcLCBroker
{
public:
    Cxc_getCodeCats();
    virtual bool parseCommand(CXmlDocument * pdoc);
    virtual bool execCommand();
    CXC_DECLARE_FACTORY()
};

#endif
```

```
#include "xc_getCommands.h"
#include "rs_code_cache.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getCodes)

bool Cxc_getCodes::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_code_intern      rsCodeIntern;
        Crs_code_cat         rsCodeCat;

        // get parms
        string strCatName;
        string strGroupCatName;
        string strGroupCodeName;

        getParm("cat_name", strCatName);
        getParm("gcat_name", strGroupCatName);
        getParm("gcode_name", strGroupCodeName);

        // edit parms
        if (strCatName.empty())
        {
            m_emLast.setError("\cat_name\ is a required parameter.");
            fSuccess = false;
            throw E_FAIL;
        }

        bool fSubGroup = !strGroupCatName.empty() && !strGroupCodeName.empty();
        if (!fSubGroup)
        {
            if (strGroupCatName.empty() && !strGroupCodeName.empty() ||
                !strGroupCatName.empty() && strGroupCodeName.empty())
            {
                m_emLast.setError("\gcat_name\ and \gcode_name\ must either both be "
                    "present or both be absent.");
                fSuccess = false;
                throw E_FAIL;
            }
        }

        CSdoConnection * pconn = m_pcoClient->getConnection();

        // retrieve the category id
        rsCodeCat.setActiveCommand("getCatId");
        rsCodeCat.setParm("cat_name", _variant_t(strCatName.c_str()));
        if (pconn->execute(rsCodeCat) == false)
        {
            m_emLast.setError(pconn->getLastErrorMessage());
            throw E_FAIL;
        }
        if (rsCodeCat.isEmpty())
        {
            m_emLast << "Code category \"\" << strCatName << "\" does not exist.";
            throw E_FAIL;
        }

        long lCatId = (long) rsCodeCat.getField("cat_id");
        long lSubCatId;
        long lSubCodeId;

        // if doing a sub group, get the code_id of the subgroup
        if (fSubGroup)
```

```

    {
        // get sub cat id
        rsCodeCat.setParm("cat_name", _variant_t(strGroupCatName.c_str()));
        if (pconn->execute(rsCodeCat) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw E_FAIL;
        }
        if (rsCodeCat.isEmpty())
        {
            m_emLast << "Sub code catagory \"" << strGroupCatName << "\" does not exist.";
            throw E_FAIL;
        }
        lSubCatId = (long) rsCodeCat.getField("cat_id");

        // get sub code id
        rsCodeIntern.setActiveCommand("getSubGroupId");
        rsCodeIntern.setParm("cat_id", _variant_t(lSubCatId));
        rsCodeIntern.setParm("code", _variant_t(strGroupCodeName.c_str()));
        if (pconn->execute(rsCodeIntern) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw E_FAIL;
        }
        if (rsCodeIntern.isEmpty())
        {
            m_emLast << "Group code \"" << strGroupCodeName << "\" does not exist.";
            throw E_FAIL;
        }
        lSubCodeId = (long) rsCodeIntern.getField("code_id");
    }

    // retrieve the set of codes
    if (fSubGroup)
    {
        rsCodeIntern.setActiveCommand("getSubCodeSet");
        rsCodeIntern.setParm("cat_id", _variant_t(lCatId));
        rsCodeIntern.setParm("group_code_id", _variant_t(lSubCodeId));
    }
    else
    {
        rsCodeIntern.setActiveCommand("getCodeSetByCatId");
        rsCodeIntern.setParm("cat_id", _variant_t(lCatId));
    }

    if (pconn->execute(rsCodeIntern) == false)
    {
        m_emLast.setError(pconn->getLastError());
        throw E_FAIL;
    }

    m_pdocResults = new CXmlDocument("<getCodes/>");
    fSuccess = rsCodeIntern.toXml(*m_pdocResults);
}
catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}
catch(HRESULT hrError)
{
    hrError;
    fSuccess = false;
}
catch(...)
{

```



```
        m_emLast.setError("Unkown exception raised. [Command:getCodes]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#ifndef xc_getCommands_h
#define xc_getCommands_h

#include "stdafx.h"
#include "xcLCBroker.h"

//Specific Commands Include
#include "xc_getStats.h"
#include "xc_getCodeCats.h"

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
// Declaration of most of the XML get Commands Classes.
//
// Macro derives the class from CxcLCBroker
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

DECLARE_XML_GETCMD_CLASS(Cxc_getAccountInfo)
DECLARE_XML_GETCMD_CLASS(Cxc_getAddressInfo)
DECLARE_XML_GETCMD_CLASS(Cxc_getAdmit)
DECLARE_XML_GETCMD_CLASS(Cxc_getAllergyInfo)
DECLARE_XML_GETCMD_CLASS(Cxc_getBiographicalInfo)
DECLARE_XML_GETCMD_CLASS(Cxc_getCareDirectives)
DECLARE_XML_GETCMD_CLASS(Cxc_getCholesterolReadings)
DECLARE_XML_GETCMD_CLASS(Cxc_getCodes)
DECLARE_XML_GETCMD_CLASS(Cxc_getConvertPc)
DECLARE_XML_GETCMD_CLASS(Cxc_getCurrConvertPc)
DECLARE_XML_GETCMD_CLASS(Cxc_getCurrEncounter)
DECLARE_XML_GETCMD_CLASS(Cxc_getCurrEncounterId)
DECLARE_XML_GETCMD_CLASS(Cxc_getCurrLoa)
DECLARE_XML_GETCMD_CLASS(Cxc_getCurrPreAdmit)
DECLARE_XML_GETCMD_CLASS(Cxc_getCurrTransfer)
DECLARE_XML_GETCMD_CLASS(Cxc_getDiagnosis)
DECLARE_XML_GETCMD_CLASS(Cxc_getDisability)
DECLARE_XML_GETCMD_CLASS(Cxc_getDischarge)
DECLARE_XML_GETCMD_CLASS(Cxc_getDischargeHistory)
DECLARE_XML_GETCMD_CLASS(Cxc_getEmploymentInfo)
DECLARE_XML_GETCMD_CLASS(Cxc_getEncounterTree)
DECLARE_XML_GETCMD_CLASS(Cxc_getExternalIDs)
DECLARE_XML_GETCMD_CLASS(Cxc_getFamilyTree)
DECLARE_XML_GETCMD_CLASS(Cxc_getGuarantorInfo)
DECLARE_XML_GETCMD_CLASS(Cxc_getInsuranceCoverage)
DECLARE_XML_GETCMD_CLASS(Cxc_getInsuranceInfo)
DECLARE_XML_GETCMD_CLASS(Cxc_getLoa)
DECLARE_XML_GETCMD_CLASS(Cxc_getLoaHistory)
DECLARE_XML_GETCMD_CLASS(Cxc_getMiscIDs)
DECLARE_XML_GETCMD_CLASS(Cxc_getPatientValuables)
DECLARE_XML_GETCMD_CLASS(Cxc_getPhone)
DECLARE_XML_GETCMD_CLASS(Cxc_getPhysicalInfo)
DECLARE_XML_GETCMD_CLASS(Cxc_getPhysicianInfo)
DECLARE_XML_GETCMD_CLASS(Cxc_getPreAdmit)
DECLARE_XML_GETCMD_CLASS(Cxc_getSecurityInfo)
DECLARE_XML_GETCMD_CLASS(Cxc_getTransfer)
DECLARE_XML_GETCMD_CLASS(Cxc_getCpiIdExists)
DECLARE_XML_GETCMD_CLASS(Cxc_getCodesByName)
DECLARE_XML_GETCMD_CLASS(Cxc_getCpiId)
DECLARE_XML_GETCMD_CLASS(Cxc_getPerson)
DECLARE_XML_GETCMD_CLASS(Cxc_getName)
DECLARE_XML_GETCMD_CLASS(Cxc_getNok)
DECLARE_XML_GETCMD_CLASS(Cxc_getNokAll)
DECLARE_XML_GETCMD_CLASS(Cxc_getCompany)
DECLARE_XML_GETCMD_CLASS(Cxc_getNewEncounterId)
DECLARE_XML_GETCMD_CLASS(Cxc_getPhysicians)
DECLARE_XML_GETCMD_CLASS(Cxc_getFacilities)
DECLARE_XML_GETCMD_CLASS(Cxc_getPocs)
```

```
DECLARE_XML_GETCMD_CLASS(Cxc_getRooms)
DECLARE_XML_GETCMD_CLASS(Cxc_getBeds)
DECLARE_XML_GETCMD_CLASS(Cxc_getInsPlans)
DECLARE_XML_GETCMD_CLASS(Cxc_getInsPlansByCompany)
DECLARE_XML_GETCMD_CLASS(Cxc_getPatientStatus)
DECLARE_XML_GETCMD_CLASS(Cxc_getPatientLocation)
DECLARE_XML_GETCMD_CLASS(Cxc_getInPatients)
DECLARE_XML_GETCMD_CLASS(Cxc_getPasswordReminder)
DECLARE_XML_GETCMD_CLASS(Cxc_getIdealBPRanges)
DECLARE_XML_GETCMD_CLASS(Cxc_getBloodPressureReadings)
DECLARE_XML_GETCMD_CLASS(Cxc_getPulseReadings)
DECLARE_XML_GETCMD_CLASS(Cxc_getWeightReadings)
DECLARE_XML_GETCMD_CLASS(Cxc_getSLMDLocations)
DECLARE_XML_GETCMD_CLASS(Cxc_getUserBiographics)
DECLARE_XML_GETCMD_CLASS(Cxc_getUserPhysicians)
DECLARE_XML_GETCMD_CLASS(Cxc_getUserInsurance)
DECLARE_XML_GETCMD_CLASS(Cxc_getHealthConditions)
DECLARE_XML_GETCMD_CLASS(Cxc_getImmunizations)
DECLARE_XML_GETCMD_CLASS(Cxc_getMedications)
DECLARE_XML_GETCMD_CLASS(Cxc_getSurgeryInfo)
DECLARE_XML_GETCMD_CLASS(Cxc_getTherapyInfo)
DECLARE_XML_GETCMD_CLASS(Cxc_getFamilyHistory)
DECLARE_XML_GETCMD_CLASS(Cxc_getImagingInfo)
DECLARE_XML_GETCMD_CLASS(Cxc_getReminder)
DECLARE_XML_GETCMD_CLASS(Cxc_getMassMailing)
DECLARE_XML_GETCMD_CLASS(Cxc_getNewUnregUserId)
DECLARE_XML_GETCMD_CLASS(Cxc_getLifeclinicStats)
DECLARE_XML_GETCMD_CLASS(Cxc_getUserPreference)
```

```
#endif
```

```
#include "xc_getCommands.h"
#include "rs_company.h"
```

```
////////////////////////////////////
```

```
CXC_IMPLEMENT_FACTORY(Cxc_getCompany)
```

```
bool Cxc_getCompany::execCommand()
```

```
{
    bool fSuccess = false;

    try
    {
        Crs_company rsCompany;

        rsCompany.setActiveCommand("cmdFetch");

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsCompany)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getCompany/>");
            fSuccess = rsCompany.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getCompany]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_convert_pc.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getConvertPc)

bool Cxc_getConvertPc::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_convert_pc  rsConvertPc;

        string strEncId;
        string strRecId;
        long lEncId = 0;

        //set the command
        rsConvertPc.setActiveCommand("cmdFetch");

        //get the parameter
        bool fParamExist = getParm("enc_id", strEncId);
        if (fParamExist) lEncId = atol(strEncId.c_str());

        if (!fParamExist || lEncId == 0)
        {
            m_emLast.setError("\enc_id\" is a required parameter and should not be 0.");
            return false;
        }

        //set the parameter
        rsConvertPc.setParameter("enc_id", _variant_t(lEncId));

        //Do we have the rec_id.....? If yes, then set rec_id parameter.
        if (getParm("rec_id", strRecId) == true)
        {
            long lRecId = atol(strRecId.c_str());
            if (lRecId) rsConvertPc.setParameter("rec_id", _variant_t(lRecId));
        }

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsConvertPc)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getConvertPc/>");
            fSuccess = rsConvertPc.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getConvertPc]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_convert_pc.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getCurrConvertPc)

bool Cxc_getCurrConvertPc::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_convert_pc rsConvertPc;
        string strEncId;

        rsConvertPc.setActiveCommand("cmdFetchCurrent");

        if (getParam("enc_id", strEncId) == false)
        {
            m_emLast.setError("\"enc_id\" is a required parameter.");
            return false;
        }

        long lEncId = atol(strEncId.c_str());
        rsConvertPc.setParameter("enc_id", _variant_t(lEncId));

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsConvertPc)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getCurrConvertPc/>");
            fSuccess = rsConvertPc.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getCurrConvertPc]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_encounter.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getCurrEncounter)

////////////////////////////////////
// Get the current Encounter from Cpi_id.
////////////////////////////////////

bool Cxc_getCurrEncounter::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_encounter rsEncId;
        string strCpiId;

        rsEncId.setActiveCommand("cmdFetchCurrentEncounter");

        if (getParm("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\"cpi_id\" is a required parameter.");
            return false;
        }

        _variant_t vCpiID(atol(strCpiId.c_str()));
        rsEncId.setParameter("cpi_id", _variant_t(vCpiID));

        CSdoConnection *pconn = m_pcoClient->getConnection();
        if ((fSuccess = pconn->execute(rsEncId)) == false)
        {
            m_emLast.setError(pconn->getLastError());
        }
        else
        {
            m_pdocResults = new CXmlDocument("<getCurrEncounter/>");
            fSuccess = rsEncId.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getCurrEncounter]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_encounter.h"
```

```
////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getCurrEncounterId)
```

```
////////////////////////////////////////
// Get the current Encounter from Cpi_id.
////////////////////////////////////////
```

```
bool Cxc_getCurrEncounterId::execCommand()
{
```

```
    bool fSuccess = false;
```

```
    try
```

```
    {
```

```
        Crs_encounter rsEncId;
        string strCpiId;
```

```
        rsEncId.setActiveCommand("cmdFetchCurrentId");
```

```
        if (getParam("cpi_id", strCpiId) == false)
```

```
        {
```

```
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }
```

```
        _variant_t vCpiID(atol(strCpiId.c_str()));
```

```
        rsEncId.setParameter("cpi_id", _variant_t(vCpiID));
```

```
        CSdoConnection *pconn = m_pcoClient->getConnection();
```

```
        if ((fSuccess = pconn->execute(rsEncId)) == false)
```

```
        {
```

```
            m_emLast.setError(pconn->getLastError());
        }
```

```
        else
```

```
        {
```

```
            m_pdocResults = new CXmlDocument("<getCurrEncounterId/>");
            fSuccess = rsEncId.toXml(*m_pdocResults);
        }
```

```
    }
```

```
    catch(_com_error & e)
```

```
    {
```

```
        m_emLast.setError(e);
        fSuccess = false;
    }
```

```
    catch(...)
```

```
    {
```

```
        m_emLast.setError("Unknown exception raised.");
        fSuccess = false;
    }
```

```
    return fSuccess;
```

```
}
```



```
#include "xc_getCommands.h"
#include "rs_loa.h"
```

```
////////////////////////////////////
```

```
CXC_IMPLEMENT_FACTORY(Cxc_getCurrLoa)
```

```
bool Cxc_getCurrLoa::execCommand()
```

```
{
    bool fSuccess = false;

    try
    {
        Crs_loa rsLoa;
        string strEncId;

        rsLoa.setActiveCommand("cmdFetchCurrent");

        if (getParam("enc_id", strEncId) == false)
        {
            m_emLast.setError("\enc_id\" is a required parameter.");
            return false;
        }

        long lEncId = atol(strEncId.c_str());
        rsLoa.setParameter("enc_id", _variant_t(lEncId));

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsLoa)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getCurrLoa/>");
            fSuccess = rsLoa.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getCurrLoa]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_pre_admit.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getCurrPreAdmit)

bool Cxc_getCurrPreAdmit::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_pre_admit rsPreAdmit;
        string strEncId;

        rsPreAdmit.setActiveCommand("cmdFetchCurrent");

        if (getParam("enc_id", strEncId) == false)
        {
            m_emLast.setError("\enc_id\" is a required parameter.");
            return false;
        }

        long lEncId = atol(strEncId.c_str());
        rsPreAdmit.setParameter("enc_id", _variant_t(lEncId));

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsPreAdmit)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getCurrPreAdmit/>");
            fSuccess = rsPreAdmit.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getCurrPreAdtmit]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_transfer.h"
```

```
////////////////////////////////////
```

```
CXC_IMPLEMENT_FACTORY(Cxc_getCurrTransfer)
```

```
bool Cxc_getCurrTransfer::execCommand()
```

```
{
    bool fSuccess = false;

    try
    {
        Crs_transfer rsTransfer;
        string strEncId;

        rsTransfer.setActiveCommand("cmdFetchCurrent");

        if (getParm("enc_id", strEncId) == false)
        {
            m_emLast.setError("\\"enc_id\\" is a required parameter.");
            return false;
        }

        long lEncId = atol(strEncId.c_str());
        rsTransfer.setParameter("enc_id", _variant_t(lEncId));

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsTransfer)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getCurrTransfer/>");
            fSuccess = rsTransfer.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getCurrTransfer]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_diagnosis.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getDiagnosis)

bool Cxc_getDiagnosis::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_diagnosis    rsDiagnosis;

        string strEncId;
        string strRecId;

        rsDiagnosis.setActiveCommand("cmdFetch");

        if (getParm("enc_id", strEncId) == false)
        {
            m_emLast.setError("\enc_id\" is a required parameter.");
            return false;
        }
        long lEncId = atol(strEncId.c_str());
        rsDiagnosis.setParameter("enc_id", _variant_t(lEncId));

        if (getParm("rec_id", strRecId) == true)
        {
            long lRecId = atol(strRecId.c_str());
            rsDiagnosis.setParameter("rec_id", _variant_t(lRecId));
        }

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsDiagnosis)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getDiagnosis/>");
            fSuccess = rsDiagnosis.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getDiagnosis]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_disability.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getDisability)

bool Cxc_getDisability::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_disability rsDisability;
        string strCpiID;

        rsDisability.setActiveCommand("cmdFetch");

        if (getParam("cpi_id", strCpiID) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }

        _variant_t vCpiID(atol(strCpiID.c_str()));
        rsDisability.setParameter("cpi_id", vCpiID);

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsDisability)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getDisability/>");
            fSuccess = rsDisability.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getDisability]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_discharge.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getDischarge)

bool Cxc_getDischarge::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_discharge rsDischarge;
        string strEncId;

        rsDischarge.setActiveCommand("cmdFetch");

        if (getParam("enc_id", strEncId) == false)
        {
            m_emLast.setError("\\"enc_id\\" is a required parameter.");
            return false;
        }

        long lEncId = atol(strEncId.c_str());
        rsDischarge.setParameter("enc_id", _variant_t(lEncId));

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsDischarge)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getDischarge/>");
            fSuccess = rsDischarge.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getDischarge]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_discharge.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getDischargeHistory)

bool Cxc_getDischargeHistory::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_discharge rsDischarge;
        string strCpiID;

        rsDischarge.setActiveCommand("cmdFetchAll");

        if (getParam("cpi_id", strCpiID) == false)
        {
            m_emLast.setError("\\"cpi_id\\" is a required parameter.");
            return false;
        }

        _variant_t vCpiID(atol(strCpiID.c_str()));
        rsDischarge.setParameter("cpi_id", vCpiID);

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsDischarge)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getDischargeHistory/>");
            fSuccess = rsDischarge.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getDischargeHistory]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_employers.h"

////////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getEmploymentInfo)

bool Cxc_getEmploymentInfo::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_employers    rsEmployer;
        string strCpiID;
        string strActiveSw;

        rsEmployer.setActiveCommand("cmdFetch");

        if (getParam("cpi_id", strCpiID) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }

        if (strCpiID.empty())
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            return false;
        }

        _variant_t vCpiID(atol(strCpiID.c_str()));
        rsEmployer.setParameter("cpi_id", vCpiID);

        if (getParam("active_sw", strActiveSw) == true)
        {
            _variant_t vActiveSw(atol(strActiveSw.c_str()));
            rsEmployer.setParameter("active_sw", vActiveSw);
        }

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsEmployer)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getEmploymentInfo/>");
            fSuccess = rsEmployer.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getEmploymentInfo]");
        fSuccess = false;
    }

    return fSuccess;
}
```



```
#include "xc_getCommands.h"
#include "rs_encounter_tree.h"

/*****

Function : Returns all encounters by the CpiId.

*****/

CXC_IMPLEMENT_FACTORY(Cxc_getEncounterTree)

bool Cxc_getEncounterTree::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_encounter_tree rsEncounterTree;
        string strCpiId;
        string strEncId;
        long lCpiId = 0;

        rsEncounterTree.setActiveCommand("cmdFetch");

        //get the parameter
        bool fParamExist = getParm("cpi_id", strCpiId);
        if (fParamExist) lCpiId = atol(strCpiId.c_str());

        if (!fParamExist || lCpiId == 0)
        {
            m_emLast.setError("\\"cpi_id\\" is a required parameter and should not be 0.");
            return false;
        }

        //set the paramter.
        rsEncounterTree.setParameter("cpi_id", _variant_t(lCpiId));

        // Currently the Stored Procedure doesn't do anything with the enc_id

        //Do we have the enc_id.....? If yes, then set enc_id parameter.
        if (getParm("enc_id", strEncId) == true)
        {
            long lEncId = atol(strEncId.c_str());
            if (lEncId) rsEncounterTree.setParameter("enc_id", _variant_t(lEncId));
        }

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsEncounterTree)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getEncounterTree/>");
            fSuccess = rsEncounterTree.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getEncounterTree]");
        fSuccess = false;
    }
}
```

```
    return fSuccess;  
}
```

```
#include "xc_getCommands.h"
#include "rs_id_map.h"

////////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getExternalIDs)

bool Cxc_getExternalIDs::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_id_map rsExternalIDs;
        string strParm;

        rsExternalIDs.setActiveCommand("cmdFetch");

        if (getParam("cpi_id", strParm) == false)
        {
            m_emLast.setError("\cpi_id\ is a required parameter.");
            return false;
        }
        _variant_t vCpiID(atol(strParm.c_str()));
        rsExternalIDs.setParameter("cpi_id", vCpiID);

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsExternalIDs)) == false)
            m_emLast.setError(pconn->getLastErrorMessage());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getExternalIDs/>");
            fSuccess = rsExternalIDs.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getExternalIDs]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_sys_org_facility.h"

////////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getFacilities)

bool Cxc_getFacilities::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_facility rs_facility;

        rs_facility.setActiveCommand("cmdFetchAll");

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rs_facility)) == false)
            m_emLast.setError(pconn->getLastErrorMessage());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getFacilities/>");
            fSuccess = rs_facility.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getFacilities]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_GetCommands.h"
#include "rs_family_history.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getFamilyHistory)

bool Cxc_getFamilyHistory::execCommand()
{
    bool fSuccess = true;

    try
    {
        Crs_family_history rs_family_history;
        string strCpiId;

        CSdoConnection * pconn = m_pcoClient->getConnection();

        //get cpi_id
        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            throw fSuccess = false;
        }

        //check if cpi_id is null.
        if (strCpiId.empty())
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        long lCpiId = atol(strCpiId.c_str());

        //fetch all bp records.
        rs_family_history.clearParms();
        rs_family_history.setRecordSetToNull();
        rs_family_history.setActiveCommand("cmdFetch");
        rs_family_history.setParm("cpi_id", _variant_t (lCpiId));
        if (pconn->execute(rs_family_history) == false)
        {
            m_emLast.setError(pconn->getLastErrorMessage());
            throw fSuccess = false;
        }
        else
        {
            m_pdocResults = new CXmlDocument("<getFamilyHistory/>");
            fSuccess = rs_family_history.toXml(*m_pdocResults);
        }
    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getFamilyHistory]");
        fSuccess = false;
    }
}
```

```
    return fSuccess;  
}
```

```
#include "xc_getCommands.h"
#include "rs_family_tree.h"
#include "do_family_tree.h"
```

```
////////////////////////////////////
```

```
CXC_IMPLEMENT_FACTORY(Cxc_getFamilyTree)
```

```
bool Cxc_getFamilyTree::execCommand()
```

```
{
    bool fSuccess = false;

    try
    {
        CDoFamilyTree    doFamilyTree;

        string strCpiID;
        getParm("cpi_id", doFamilyTree.m_strCpiID);

        if (fSuccess = doFamilyTree.fromConnection(m_pcoClient->getConnection()))
        {
            m_pdocResults = new CXmlDocument("<getFamilyTree/>");
            fSuccess = doFamilyTree.toXml(*m_pdocResults);
        }
        else
            m_emLast.setError(doFamilyTree.getLastErrorMessage());
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getFamilyTree]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_guarantor.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getGuarantorInfo)

bool Cxc_getGuarantorInfo::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_guarantor rsGuarantor;
        string strCpiID;

        rsGuarantor.setActiveCommand("cmdFetch");

        if (getParm("cpi_id", strCpiID) == false)
        {
            m_emLast.setError("\\"cpi_id\\" is a required parameter.");
            return false;
        }
        _variant_t vCpiID(atol(strCpiID.c_str()));
        rsGuarantor.setParameter("cpi_id", vCpiID);

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsGuarantor)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getGuarantorInfo/>");
            fSuccess = rsGuarantor.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getGuarantorInfo]");
        fSuccess = false;
    }

    return fSuccess;
}
```



```
#include "xc_GetCommands.h"
#include "rs_health_condition.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getHealthConditions)

bool Cxc_getHealthConditions::execCommand()
{
    bool fSuccess = true;

    try
    {
        Crs_health_condition rs_hc;
        string strCpiId;

        CSdoConnection * pconn = m_pcoClient->getConnection();

        //get cpi_id
        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            throw fSuccess = false;
        }

        //check if cpi_id is null.
        if (strCpiId.empty())
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        long lCpiId = atol(strCpiId.c_str());

        //fetch all bp records.
        rs_hc.clearParms();
        rs_hc.setRecordSetToNull();
        rs_hc.setActiveCommand("cmdFetch");
        rs_hc.setParm("cpi_id", _variant_t (lCpiId));
        if (pconn->execute(rs_hc) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
        else
        {
            m_pdocResults = new CXmlDocument("<getHealthConditions/>");
            fSuccess = rs_hc.toXml(*m_pdocResults);
        }
    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getHealthConditions]");
        fSuccess = false;
    }
}
```

```
    return fSuccess;  
}
```

```

#include "xc_GetCommands.h"
#include "rs_code_cache.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getIdealBPRanges)

bool Cxc_getIdealBPRanges::execCommand()
{
    bool fSuccess = true;

    try
    {
        Crs_code_intern rsCode;
        string strCatName;

        CSdoConnection * pconn = m_pcoClient->getConnection();

        //Fixed category name
        strCatName = "Blood Pressure Range";

        rsCode.setActiveCommand("getSortedCodeSetByCatName");
        rsCode.setParm("cat_name", _variant_t (strCatName.c_str()));

        if (pconn->execute(rsCode) == false)
        {
            m_emLast.setError(pconn->getLastErrorMessage());
            throw fSuccess = false;
        }

        //check if values present
        if (rsCode.isEmpty())
        {
            m_emLast << "Ideal Blood Pressure values not present";
            throw fSuccess = false;
        }

        //extract the values.

        string strCode, strValue;
        string strSysLow, strSysHigh, strDiasLow, strDiasHigh;

        while (!rsCode.isEOF())
        {
            rsCode.getField("code", strCode);
            rsCode.getField("code_name", strValue);

            if (strCode == "SYS_LOW")
                strSysLow = strValue;
            if (strCode == "SYS_HIGH")
                strSysHigh = strValue;
            if (strCode == "DIAS_LOW")
                strDiasLow = strValue;
            if (strCode == "DIAS_HIGH")
                strDiasHigh = strValue;

            //forward the recordset.
            rsCode.getADO()->MoveNext();
        }

        //construct the result document.

        m_pdocResults = new CXmlDocument("<getIdealBPRanges/>");
        openXmlTag("row", XML_TYPE_ROW);
        addXmlChild("lowsystolic", strSysLow.c_str());
    }
}

```

```
        addXmlChild("highsystolic", strSysHigh.c_str());
        addXmlChild("lowdiastolic", strDiasLow.c_str());
        addXmlChild("highdiastolic", strDiasHigh.c_str());

        closeXmlTag();

    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getIdealBPInfo]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_GetCommands.h"
#include "rs_imaging.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getImagingInfo)

bool Cxc_getImagingInfo::execCommand()
{
    bool fSuccess = true;

    try
    {
        Crs_imaging rs_imaging;
        string strCpiId;

        CSdoConnection * pconn = m_pcoClient->getConnection();

        //get cpi_id
        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            throw fSuccess = false;
        }

        //check if cpi_id is null.
        if (strCpiId.empty())
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        long lCpiId = atol(strCpiId.c_str());

        //fetch all bp records.
        rs_imaging.clearParms();
        rs_imaging.setRecordSetToNull();
        rs_imaging.setActiveCommand("cmdFetch");
        rs_imaging.setParm("cpi_id", _variant_t (lCpiId));
        if (pconn->execute(rs_imaging) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
        else
        {
            m_pdocResults = new CXmlDocument("<getImagingInfo/>");
            fSuccess = rs_imaging.toXml(*m_pdocResults);
        }

    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getImagingInfo]");
        fSuccess = false;
    }
}
```

```
    return fSuccess;  
}
```

```
#include "xc_GetCommands.h"
#include "rs_immunization.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getImmunizations)

bool Cxc_getImmunizations::execCommand()
{
    bool fSuccess = true;

    try
    {
        Crs_immunization rs_immunization;
        string strCpiId;

        CSdoConnection * pconn = m_pcoClient->getConnection();

        //get cpi_id
        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            throw fSuccess = false;
        }

        //check if cpi_id is null.
        if (strCpiId.empty())
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        long lCpiId = atol(strCpiId.c_str());

        //fetch all bp records.
        rs_immunization.clearParms();
        rs_immunization.setRecordSetToNull();
        rs_immunization.setActiveCommand("cmdFetch");
        rs_immunization.setParm("cpi_id", _variant_t (lCpiId));
        if (pconn->execute(rs_immunization) == false)
        {
            m_emLast.setError(pconn->getLastErrorMessage());
            throw fSuccess = false;
        }
        else
        {
            m_pdocResults = new CXmlDocument("<getImmunizations/>");
            fSuccess = rs_immunization.toXml(*m_pdocResults);
        }
    }

    catch(bool fError)
    {
        fError;
    }

    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }

    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getImmunizations]");
        fSuccess = false;
    }
}
```

```
    return fSuccess;  
}
```



```
#include "xc_getCommands.h"
#include "rs_patient.h"
```

```
////////////////////////////////////
```

```
CXC_IMPLEMENT_FACTORY(Cxc_getInPatients)
```

```
bool Cxc_getInPatients::execCommand()
```

```
{
```

```
    bool fSuccess = false;
```

```
    try
```

```
    {
```

```
        Crs_patient rs_patient;
```

```
        rs_patient.setActiveCommand("cmdFetchAll");
```

```
        CSdoConnection * pconn = m_pcoClient->getConnection();
```

```
        if ((fSuccess = pconn->execute(rs_patient)) == false)
            m_emLast.setError(pconn->getLastError());
```

```
        if (fSuccess)
```

```
        {
```

```
            m_pdocResults = new CXmlDocument("<getInPatients/>");
```

```
            fSuccess = rs_patient.toXml(*m_pdocResults);
```

```
        }
```

```
    }
    catch(_com_error & e)
```

```
    {
```

```
        m_emLast.setError(e);
```

```
        fSuccess = false;
```

```
    }
```

```
    catch(...)
```

```
    {
```

```
        m_emLast.setError("Unknown exception raised. [Command:getInPatients]");
```

```
        fSuccess = false;
```

```
    }
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_getCommands.h"
#include "rs_insurance.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getInsPlans)

bool Cxc_getInsPlans::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_insurance_plan rs_insurance_plan;

        rs_insurance_plan.setActiveCommand("cmdFetchAll");

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rs_insurance_plan)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getInsPlans/>");
            fSuccess = rs_insurance_plan.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getInsPlans]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_insurance.h"
```

```
////////////////////////////////////
```

```
CXC_IMPLEMENT_FACTORY(Cxc_getInsPlansByCompany)
```

```
bool Cxc_getInsPlansByCompany::execCommand()
```

```
{
    bool fSuccess = false;

    try
    {
        Crs_insurance_plan rs_insurance_plan;
        string strCompId;

        rs_insurance_plan.setActiveCommand("cmdFetchByCompany");

        if ( getParm("ins_company_id", strCompId) == false )
        {
            m_emLast.setError("\\"ins_company_id\\" is a required parameter.");
            return false;
        }

        _variant_t vCompID(atol(strCompId.c_str()));
        rs_insurance_plan.setParameter("ins_company_id", _variant_t(vCompID));

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rs_insurance_plan)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getInsPlansByCompany/>");
            fSuccess = rs_insurance_plan.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getInsPlansByCompany]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_insurance.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getInsuranceCoverage)

bool Cxc_getInsuranceCoverage::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_insurance rsIns;
        string strCpiID;
        string strEncID;
        string strActiveSw;

        rsIns.setActiveCommand("cmdFetchInsuranceCoverage");

        if (getParm("cpi_id", strCpiID) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }
        _variant_t vCpiID(atol(strCpiID.c_str()));
        rsIns.setParameter("cpi_id", vCpiID);

        if (getParm("enc_id", strEncID) == false)
        {
            m_emLast.setError("\enc_id\" is a required parameter.");
            return false;
        }
        _variant_t vEncID(atol(strEncID.c_str()));
        rsIns.setParameter("enc_id", vEncID);

        if (getParm("active_sw", strActiveSw) == true)
        {
            _variant_t vActiveSw(atol(strActiveSw.c_str()));
            rsIns.setParameter("active_sw", vActiveSw);
        }

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsIns)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getInsuranceCoverage/>");
            fSuccess = rsIns.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getInsuranceCoverage]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_insurance.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getInsuranceInfo)

bool Cxc_getInsuranceInfo::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_insurance rsInsurance;
        string strCpiID;
        string strActiveSw;

        rsInsurance.setActiveCommand("cmdFetch");

        if (getParam("cpi_id", strCpiID) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }
        _variant_t vCpiID(atol(strCpiID.c_str()));
        rsInsurance.setParameter("cpi_id", vCpiID);

        if (getParam("active_sw", strActiveSw) == true)
        {
            _variant_t vActiveSw(atol(strActiveSw.c_str()));
            rsInsurance.setParameter("active_sw", vActiveSw);
        }

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsInsurance)) == false)
            m_emLast.setError(pconn->getLastErrorMessage());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getInsuranceInfo/>");
            fSuccess = rsInsurance.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getInsuranceInfo]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_stats.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getLifeclinicStats)

bool Cxc_getLifeclinicStats::execCommand()
{
    bool fSuccess = true;

    try
    {
        Crs_stats rs_stats;

        CSdoConnection * pconn = m_pcoClient->getConnection();

        rs_stats.setActiveCommand("cmdFetchStats");
        if ((fSuccess = pconn->execute(rs_stats)) == false)
        {
            m_emLast.setError(pconn->getLastErrorMessage());
            throw fSuccess = false;
        }

        //return XML results.
        m_pdocResults = new CXmlDocument("<getLifeclinicStats/>");
        fSuccess = rs_stats.toXml(*m_pdocResults);

    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getLifeclinicStats]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_loa.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getLoa)

bool Cxc_getLoa::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_loa rsLoa;
        string strEncId;
        string strRecId;
        long lEncId = 0;

        //set the command
        rsLoa.setActiveCommand("cmdFetch");

        //get the parameter
        bool fParamExist = getParm("enc_id", strEncId);
        if (fParamExist) lEncId = atol(strEncId.c_str());

        if (!fParamExist || lEncId == 0)
        {
            m_emLast.setError("\"enc_id\" is a required parameter and should not be 0.");
            return false;
        }

        //set the parameter
        rsLoa.setParameter("enc_id", _variant_t(lEncId));

        //Do we have the rec_id.....? If yes, then set rec_id parameter.
        if (getParm("rec_id", strRecId) == true)
        {
            long lRecId = atol(strRecId.c_str());
            if (lRecId) rsLoa.setParameter("rec_id", _variant_t(lRecId));
        }

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsLoa)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getLoa/>");
            fSuccess = rsLoa.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getLoa]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_loa.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getLoaHistory)

bool Cxc_getLoaHistory::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_loa rsLoa;
        string strCpiID;

        rsLoa.setActiveCommand("cmdFetchAll");

        if (getParm("cpi_id", strCpiID) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }

        _variant_t vCpiID(atol(strCpiID.c_str()));
        rsLoa.setParameter("cpi_id", vCpiID);

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsLoa)) == false)
            m_emLast.setError(pconn->getLastErrorMessage());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getLoaHistory/>");
            fSuccess = rsLoa.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getLoaHistory]");
        fSuccess = false;
    }

    return fSuccess;
}
```



```
#include "xc_getCommands.h"
#include "rs_mass_mailing.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getMassMailing)

bool Cxc_getMassMailing::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_mass_mailing      rsMassMailing;

        string strCpiId;
        string strRecId;

        getParm("cpi_id", strCpiId);
        getParm("rec_id", strRecId);

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if (strRecId.size())
        {
            rsMassMailing.setActiveCommand("cmdGetByRecId");
            rsMassMailing.setParameter("rec_id", _variant_t(atol(strRecId.c_str())));
        }
        else if (strCpiId.size())
        {
            rsMassMailing.setActiveCommand("cmdGetByCpiId");
            rsMassMailing.setParameter("cpi_id", _variant_t(atol(strCpiId.c_str())));
        }
        else
            rsMassMailing.setActiveCommand("cmdGetAll");

        if ((fSuccess = pconn->execute(rsMassMailing)) == false)
        {
            m_emLast.setError(pconn->getLastError());
        }
        else
        {
            m_pdocResults = new CXmlDocument("<getMassMailing/>");
            fSuccess = rsMassMailing.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getReminder]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_GetCommands.h"
#include "rs_medication.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getMedications)

bool Cxc_getMedications::execCommand()
{
    bool fSuccess = true;

    try
    {
        Crs_medication rs_medication;
        string strCpiId;

        CSdoConnection * pconn = m_pcoClient->getConnection();

        //get cpi_id
        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            throw fSuccess = false;
        }

        //check if cpi_id is null.
        if (strCpiId.empty())
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        long lCpiId = atol(strCpiId.c_str());

        //fetch all bp records.
        rs_medication.clearParms();
        rs_medication.setRecordSetToNull();
        rs_medication.setActiveCommand("cmdFetch");
        rs_medication.setParm("cpi_id", _variant_t (lCpiId));
        if (pconn->execute(rs_medication) == false)
        {
            m_emLast.setError(pconn->getLastErrorMessage());
            throw fSuccess = false;
        }
        else
        {
            m_pdocResults = new CXmlDocument("<getMedications/>");
            fSuccess = rs_medication.toXml(*m_pdocResults);
        }
    }

    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getMedications]");
        fSuccess = false;
    }
}
```

```
    return fSuccess;  
}
```

```
#include "xc_getCommands.h"
#include "rs_misc_id.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getMiscIDs)

bool Cxc_getMiscIDs::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_misc_id rsMiscIDs;
        string strParm;

        rsMiscIDs.setActiveCommand("cmdFetch");

        if (getParam("cpi_id", strParm) == false)
        {
            m_emLast.setError("\\"cpi_id\\" is a required parameter.");
            return false;
        }
        _variant_t vCpiID(atol(strParm.c_str()));
        rsMiscIDs.setParameter("cpi_id", vCpiID);

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsMiscIDs)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getMiscIDs/>");
            fSuccess = rsMiscIDs.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getMiscIDs]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_name.h"
```

```
////////////////////////////////////
```

```
CXC_IMPLEMENT_FACTORY(Cxc_getName)
```

```
bool Cxc_getName::execCommand()
```

```
{
    bool fSuccess = false;

    try
    {
        Crs_name      rsName;

        string strCpiID;
        string strActiveSw;

        if (getParam("cpi_id", strCpiID) == false)
        {
            m_emLast.setError("\cpi_id\ is a required parameter.");
            return false;
        }
        _variant_t vCpiID(atol(strCpiID.c_str()));

        rsName.setActiveCommand("cmdFetch");
        rsName.setParameter("cpi_id", vCpiID);

        if (getParam("active_sw", strActiveSw) == true)
        {
            _variant_t vActiveSw(atol(strActiveSw.c_str()));
            rsName.setParameter("active_sw", vActiveSw);
        }

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsName)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getName/>");
            fSuccess = rsName.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getName]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_encounter.h"

////////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getNewEncounterId)

bool Cxc_getNewEncounterId::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_encounter    rsEncounter;

        //set the command
        rsEncounter.setActiveCommand("cmdFetchNewEncounterId");

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsEncounter)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getLoa/>");
            fSuccess = rsEncounter.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getNewEncounterId]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"

#include "rs_unregistered_user.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getNewUnregUserId)

bool Cxc_getNewUnregUserId::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_unregistered_user rsUnregUser;

        //get db connection
        CSdoConnection * pconn = m_pcoClient->getConnection();

        //get new id.
        long lUnregUserId = getNewUnregUserId();
        if (!lUnregUserId)
        {
            m_emLast.setError("Unexpected error. Could not get new unregistered user id.")✗
            throw fSuccess = false;
        }
        long lAuditId = getAuditId();
        if (!lAuditId)
        {
            m_emLast.setError("Unexpected error. Could not get new audit id.");
            throw fSuccess = false;
        }

        //////////////////////////////////////
        // update the access date for the unreg user.
        //////////////////////////////////////

        DATE dtEffectiveDate;
        dtEffectiveDate = (DATE) COleDateTime::GetCurrentTime();

        //update the unreg user
        rsUnregUser.clearParms();
        rsUnregUser.setRecordSetToNull();
        rsUnregUser.setActiveCommand("cmdUpdate");
        rsUnregUser.setParameter("user_id", _variant_t(lUnregUserId));
        rsUnregUser.setParameter("effective_dt", _variant_t(dtEffectiveDate));
        rsUnregUser.setParameter("access_dt", _variant_t(dtEffectiveDate));
        rsUnregUser.setParameter("audit_id", _variant_t(lAuditId));
        if ((fSuccess = pconn->execute(rsUnregUser)) == false)
        {
            m_emLast.setError(pconn->getLastErrorMessage());
            throw fSuccess = false;
        }

        //construct the XML result.
        m_pdocResults = new CXmlDocument("<getNewUnregUserId/>");
        m_pdocResults->addChild("user_id", _variant_t(lUnregUserId));
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {

```

```
        m_emLast.setError("Unknown exception raised. [Command:getNewUnregUserId]");
        fSuccess = false;
    }

    return fSuccess;
}
```



```
#include "xc_getCommands.h"
#include "rs_nok.h"
```

```
////////////////////////////////////
```

```
CXC_IMPLEMENT_FACTORY(Cxc_getNok)
```

```
bool Cxc_getNok::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_nok rsNok;
        string strCpiId, strFName, strLName, strRelation;

        rsNok.setActiveCommand("cmdFetch");

        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\"cpi_id\" is a required parameter.");
            return false;
        }
        if (getParam("first_name", strFName) == false)
        {
            m_emLast.setError("\"first_name\" is a required parameter.");
            return false;
        }
        if (getParam("last_name", strLName) == false)
        {
            m_emLast.setError("\"last_name\" is a required parameter.");
            return false;
        }
        if (getParam("relationship", strRelation) == false)
        {
            m_emLast.setError("\"last_name\" is a required parameter.");
            return false;
        }

        long lCpiId = atol(strCpiId.c_str());
        rsNok.setParameter("cpi_id", _variant_t(lCpiId));
        rsNok.setParameter("first_name", _variant_t(strFName.c_str()));
        rsNok.setParameter("last_name", _variant_t(strLName.c_str()));
        rsNok.setParameter("relationship", _variant_t(strRelation.c_str()));

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsNok)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getNok/>");
            fSuccess = rsNok.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getNok]");
    }
}
```

```
        fSuccess = false;
    }
    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_nok.h"
```

```
////////////////////////////////////
```

```
CXC_IMPLEMENT_FACTORY(Cxc_getNokAll)
```

```
bool Cxc_getNokAll::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_nok rsNok;
        string strCpiId;

        rsNok.setActiveCommand("cmdFetchAll");

        if (getParm("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\"cpi_id\" is a required parameter.");
            return false;
        }

        long lCpiId = atol(strCpiId.c_str());
        rsNok.setParameter("cpi_id", _variant_t(lCpiId));

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsNok)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getNokAll/>");
            fSuccess = rsNok.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getNokAll]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_cpi_user.h"

////////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getPasswordReminder)

bool Cxc_getPasswordReminder::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_cpi_user rs_cpi_user;
        string strUserLogin;

        if (getParm("user_login", strUserLogin) == false)
        {
            m_emLast.setError("\\"user_login\\" is a required parameter.");
            throw fSuccess = false;
        }

        if (strUserLogin.empty())
        {
            m_emLast.setError("\\"user_login\\" is NULL.");
            throw fSuccess = false;
        }

        CSdoConnection * pconn = m_pcoClient->getConnection();

        //check if user_login is valid.
        rs_cpi_user.clearParms();
        rs_cpi_user.setRecordSetToNull();
        rs_cpi_user.setActiveCommand("cmdCheckUser");
        rs_cpi_user.setParameter("user_login", _variant_t(strUserLogin.c_str()));
        if (pconn->execute(rs_cpi_user) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }

        if (rs_cpi_user.isEmpty())           //username exists in db??
        {
            //username not present...return error
            m_emLast.setError("Invalid User.");
            throw fSuccess = false;
        }

        //fetch password reminder and email address.
        rs_cpi_user.clearParms();
        rs_cpi_user.setRecordSetToNull();
        rs_cpi_user.setActiveCommand("cmdFetchReminder");
        rs_cpi_user.setParameter("user_login", _variant_t(strUserLogin.c_str()));
        rs_cpi_user.setParameter("purpose", _variant_t("Email"));
        if ((fSuccess = pconn->execute(rs_cpi_user)) == false)
        {
            m_emLast.setError(pconn->getLastError());
        }
        else
        {
            //extract data
            string strPassRem, strEmailAddress;
            if (!rs_cpi_user.isEmpty())
            {

```

```
        rs_cpi_user.getField("password_reminder", strPassRem);
        rs_cpi_user.getField("email_address", strEmailAddress);
    }
    else
    {
        strPassRem = "";
        strEmailAddress = "";
    }

    //construct xml result.
    m_pdocResults = new CXmlDocument("<getPasswordReminder/>");

    openXmlTag("cpi_user");
    openXmlTag("row");
    addXmlChild("password_reminder", strPassRem.c_str());
    addXmlChild("email_address", strEmailAddress.c_str());
    closeXmlTag();
    closeXmlTag();
}
}
catch(bool fError)
{
    fError;
}
catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}
catch(...)
{
    m_emLast.setError("Unknown exception raised. [Command:getPasswordReminder]");
    fSuccess = false;
}

return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_location.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getPatientLocation)

bool Cxc_getPatientLocation::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_location_occupant rs_loc_occ;
        string strEncId;

        if (getParam("enc_id", strEncId) == false)
        {
            m_emLast.setError("\enc_id\" is a required parameter !!!");
            return false;
        }

        rs_loc_occ.setActiveCommand("cmdFetch");
        rs_loc_occ.setParm("enc_id", _variant_t(atol(strEncId.c_str())));

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rs_loc_occ)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getPatientLocation/>");
            fSuccess = rs_loc_occ.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getPatientLocation]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_encounter.h"

////////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getPatientStatus)

bool Cxc_getPatientStatus::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_encounter rsEncounter;
        string strCpiId;

        rsEncounter.setActiveCommand("cmdFetchPatientStatus");

        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\"cpi_id\" is a required parameter.");
            return false;
        }

        long lCpiId = atol(strCpiId.c_str());
        rsEncounter.setParameter("cpi_id", _variant_t(lCpiId));

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsEncounter)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getPatientStatus/>");
            fSuccess = rsEncounter.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getPatientStatus]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_patient_valuables.h"

////////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getPatientValuables)

bool Cxc_getPatientValuables::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_patient_valuables rsValuables;
        string strEncId;

        rsValuables.setActiveCommand("cmdFetch");

        if (getParam("enc_id", strEncId) == false)
        {
            m_emLast.setError("\enc_id\" is a required parameter.");
            return false;
        }

        long lEncId = atol(strEncId.c_str());
        rsValuables.setParameter("enc_id", _variant_t(lEncId));

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsValuables)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getPatientValuables/>");
            fSuccess = rsValuables.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getPatientValuables]");
        fSuccess = false;
    }

    return fSuccess;
}
```



```
#include "xc_getCommands.h"
#include "rs_person.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getPerson)

bool Cxc_getPerson::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_person      rsPerson;

        string strCpiID;

        if (getParm("cpi_id", strCpiID) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }
        _variant_t vCpiID(atol(strCpiID.c_str()));

        rsPerson.setActiveCommand("cmdFetch");
        rsPerson.setParameter("cpi_id", vCpiID);

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsPerson)) == false)
            m_emLast.setError(pconn->getLastErrorMessage());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getPerson/>");
            fSuccess = rsPerson.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getPerson]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_phone.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getPhone)

bool Cxc_getPhone::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_phone      rsPhone;
        string strCpiID;
        string strActiveSw;

        rsPhone.setActiveCommand("cmdFetch");

        if (getParam("cpi_id", strCpiID) == false)
        {
            m_emLast.setError("\\"cpi_id\\" is a required parameter.");
            return false;
        }
        _variant_t vCpiID(atol(strCpiID.c_str()));
        rsPhone.setParameter("cpi_id", vCpiID);

        if (getParam("active_sw", strActiveSw) == true)
        {
            _variant_t vActiveSw(atol(strActiveSw.c_str()));
            rsPhone.setParameter("active_sw", vActiveSw);
        }

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsPhone)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getPhone/>");
            fSuccess = rsPhone.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getPhone]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_physical.h"

////////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getPhysicalInfo)

bool Cxc_getPhysicalInfo::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_physical    rsPhysical;
        string strCpiID;

        rsPhysical.setActiveCommand("cmdFetch");

        if (getParam("cpi_id", strCpiID) == false)
        {
            m_emLast.setError("\\"cpi_id\\" is a required parameter.");
            return false;
        }

        long lCpiID = atol(strCpiID.c_str());
        rsPhysical.setParameter("cpi_id", _variant_t(lCpiID));

        if ((fSuccess = m_pcoClient->getConnection()->execute(rsPhysical)) == false)
            m_emLast.setError(m_pcoClient->getConnection()->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getPhysicalInfo/>");
            fSuccess = rsPhysical.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getPhysicalInfo]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_physicians.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getPhysicianInfo)

bool Cxc_getPhysicianInfo::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_physicians rsPhysician;
        string strEncID;

        rsPhysician.setActiveCommand("cmdFetchHcpByEnc");

        if (getParam("enc_id", strEncID) == false)
        {
            m_emLast.setError("\"enc_id\" is a required parameter.");
            return false;
        }
        long lEncID = atol(strEncID.c_str());
        rsPhysician.setParameter("enc_id", _variant_t(lEncID));

        if ((fSuccess = m_pcoClient->getConnection()->execute(rsPhysician)) == false)
            m_emLast.setError(m_pcoClient->getConnection()->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getPhysicianInfo/>");
            fSuccess = rsPhysician.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getPhysicianInfo]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_physicians.h"
```

```
////////////////////////////////////
```

```
CXC_IMPLEMENT_FACTORY(Cxc_getPhysicians)
```

```
bool Cxc_getPhysicians::execCommand()
```

```
{
    bool fSuccess = false;

    try
    {
        Crs_physicians  rsPhysician;

        rsPhysician.setActiveCommand("cmdFetchAll");

        CSdoConnection *pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsPhysician)) == false)
            m_emLast.setError(m_pcoClient->getConnection()->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getPhysicians/>");
            fSuccess = rsPhysician.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getPhysicians]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_location.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getPocs)

bool Cxc_getPocs::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_poc rs_poc;

        rs_poc.setActiveCommand("cmdFetchAll");

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rs_poc)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getPocs/>");
            fSuccess = rs_poc.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getPocs]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_GetCommands.h"
#include "rs_pre_admit.h"

////////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getPreAdmit)

bool Cxc_getPreAdmit::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_pre_admit    rsPreAdmit;
        string strEncId;
        string strRecId;
        long lEncId = 0;

        //set the command
        rsPreAdmit.setActiveCommand("cmdFetch");

        //get the parameter
        bool fParamExist = getParm("enc_id", strEncId);
        if (fParamExist) lEncId = atol(strEncId.c_str());

        if (!fParamExist || lEncId == 0)
        {
            m_emLast.setError("\"enc_id\" is a required parameter and should not be 0.");
            return false;
        }

        //set the parameter
        rsPreAdmit.setParameter("enc_id", _variant_t(lEncId));

        //Do we have the rec_id.....? If yes, then set rec_id parameter.
        if (getParm("rec_id", strRecId) == true)
        {
            long lRecId = atol(strRecId.c_str());
            if (lRecId) rsPreAdmit.setParameter("rec_id", _variant_t(lRecId));
        }

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsPreAdmit)) == false)
            m_emLast.setError(pconn->getLastErrorMessage());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getPreAdmit/>");
            fSuccess = rsPreAdmit.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getPreAdmit]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_GetCommands.h"
#include "rs_blood_pressure.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getPulseReadings)

bool Cxc_getPulseReadings::execCommand()
{
    bool fSuccess = true;

    try
    {
        Crs_blood_pressure rsBP;
        string strCpiId, strStartDate, strEndDate;

        CSdoConnection * pconn = m_pcoClient->getConnection();

        //get cpi_id
        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            throw fSuccess = false;
        }

        //optional parameters.
        getParam("start_dt", strStartDate);
        getParam("end_dt", strEndDate);

        //check if cpi_id is null.
        if (strCpiId.empty())
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        long lCpiId = atol(strCpiId.c_str());

        //if start_dt is not provided, then fetch all pulse records.
        if (strStartDate.empty())
        {
            //fetch all pulse records.
            rsBP.setActiveCommand("cmdFetchPulseAll");
            rsBP.setParam("cpi_id", _variant_t (lCpiId));
        }
        else
        {
            COleDateTime oledate;
            DATE dtStartDate;

            //parse start_dt.
            oledate.ParseDateTime(strStartDate.c_str());
            dtStartDate = (DATE) oledate;
            if (dtStartDate == NULL)
            {
                m_emLast.setError("\start_dt\" is Invalid.");
                throw fSuccess = false;
            }

            if (strEndDate.empty())
            {
                //fetch all bp records.
                rsBP.setActiveCommand("cmdFetchPulseByStartDate");
                rsBP.setParam("cpi_id", _variant_t (lCpiId));
                rsBP.setParam("start_dt", _variant_t (dtStartDate));
            }
        }
    }
}
```



```
else
{
    DATE dtEndDate;

    //parse end_dt.
    oledate.ParseDateTime(strEndDate.c_str());
    dtEndDate = (DATE) oledate;
    if (dtEndDate == NULL)
    {
        m_emLast.setError("\"end_dt\" is Invalid.");
        throw fSuccess = false;
    }

    //get records from start_dt to end_dt
    rsBP.setActiveCommand("cmdFetchPulseByDateRange");
    rsBP.setParm("cpi_id", _variant_t (lCpiId));
    rsBP.setParm("start_dt", _variant_t (dtStartDate));
    rsBP.setParm("end_dt", _variant_t (dtEndDate));
}

if (pconn->execute(rsBP) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
else
{
    m_pdocResults = new CXmlDocument("<getPulseReadings/>");
    fSuccess = rsBP.toXml(*m_pdocResults);
}

}
catch(bool fError)
{
    fError;
}
catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}
catch(...)
{
    m_emLast.setError("Unkown exception raised. [Command:getPulseReadings]");
    fSuccess = false;
}

return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_reminders.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getReminder)

bool Cxc_getReminder::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_reminder      rsReminder;

        string strCpiId;
        string strRecId;

        if (getParm("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }

        getParm("rec_id", strRecId);

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if (strRecId.size())
        {
            rsReminder.setActiveCommand("cmdGetOne");
            rsReminder.setParameter("rec_id", _variant_t(atol(strRecId.c_str())));
        }
        else
        {
            rsReminder.setActiveCommand("cmdGetAll");
            rsReminder.setParameter("cpi_id", _variant_t(atol(strCpiId.c_str())));
        }

        if ((fSuccess = pconn->execute(rsReminder)) == false)
        {
            m_emLast.setError(pconn->getLastError());
        }
        else
        {
            m_pdocResults = new CXmlDocument("<getReminder/>");
            fSuccess = rsReminder.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getReminder]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_location.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getRooms)

bool Cxc_getRooms::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_room rs_room;

        rs_room.setActiveCommand("cmdFetchAll");

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rs_room)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getRooms/>");
            fSuccess = rs_room.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getRooms]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_encounter.h"
```

```
////////////////////////////////////
```

```
CXC_IMPLEMENT_FACTORY(Cxc_getSecurityInfo)
```

```
bool Cxc_getSecurityInfo::execCommand()
```

```
{
    bool fSuccess = false;

    try
    {
        Crs_encounter rsEnc;
        string strEncId;

        rsEnc.setActiveCommand("cmdFetchSecurity");

        if (getParam("enc_id", strEncId) == false)
        {
            m_emLast.setError("\"enc_id\" is a required parameter.");
            return false;
        }

        _variant_t vEncID(atol(strEncId.c_str()));
        rsEnc.setParameter("enc_id", _variant_t(vEncID));

        CSdoConnection *pconn = m_pcoClient->getConnection();
        if ((fSuccess = pconn->execute(rsEnc)) == false)
        {
            m_emLast.setError(pconn->getLastErrorMessage());
        }
        else
        {
            m_pdocResults = new CXmlDocument("<getSecurityInfo/>");
            fSuccess = rsEnc.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getSecurityInfo]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_otherCommands.h"
#include "rs_company.h"
```

```
////////////////////////////////////
```

```
CXC_IMPLEMENT_FACTORY(Cxc_getSLMDLocations)
```

```
bool Cxc_getSLMDLocations::execCommand()
```

```
{
    bool fSuccess = false;

    try
    {
        Crs_company rsCompany;
        string strZip;

        if (getParam("zip", strZip) == false)
        {
            m_emLast.setError("\\"zip\\" is a required parameter.");
            throw fSuccess = false;
        }

        if (strZip.empty())
        {
            m_emLast.setError("\\"zip\\" is null.");
            throw fSuccess = false;
        }

        rsCompany.setActiveCommand("cmdFetchLocations");
        rsCompany.setParameter("zip", _variant_t (strZip.c_str()));

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsCompany)) == false)
        {
            m_emLast.setError(pconn->getLastErrorMessage());
        }
        else
        {
            m_pdocResults = new CXmlDocument("<getCompany/>");
            fSuccess = rsCompany.toXml(*m_pdocResults);
        }

    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getSLMDLocations]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getStats.h"

////////////////////////////////////
Cxc_getStats::Cxc_getStats()
{
    m_fConnectRequired = false;
}

CXC_IMPLEMENT_FACTORY(Cxc_getStats)

bool Cxc_getStats::execCommand()
{
    bool fSuccess = true;

    try
    {
        // create xml document
        m_pdocResults = new CXmlDocument("<getStats/>");

        CXmlElement elCount;
        m_pdocResults->createElement("totclients",
            _variant_t(_Module.m_statsLCBroker.m_lTotalClients), &elCount);
        m_pdocResults->addChild(&elCount);
        m_pdocResults->createElement("currclients",
            _variant_t(_Module.m_statsLCBroker.m_lCurrentClients), &elCount);
        m_pdocResults->addChild(&elCount);
        m_pdocResults->createElement("numcmds",
            _variant_t(_Module.m_statsLCBroker.m_lCommandsProcessed), &elCount);
        m_pdocResults->addChild(&elCount);

        // send start time
        BSTR bstrDate;
        VarBstrFromDate(_Module.m_statsLCBroker.m_vartimeStarted, 0, 0, &bstrDate);
        CXmlElement elTime;
        m_pdocResults->createElement("starttime", (char *) _bstr_t(bstrDate, false), &
        elTime);
        m_pdocResults->addChild(&elTime);

        // send current time
        SYSTEMTIME st;
        GetLocalTime(&st);
        DATE timeNow;
        SystemTimeToVariantTime(&st, &timeNow);
        VarBstrFromDate(timeNow, 0, 0, &bstrDate);
        m_pdocResults->createElement("currenttime", (char *) _bstr_t(bstrDate, false), &
        elTime);
        m_pdocResults->addChild(&elTime);

        // this command should not be included in the total command processed count
        // as this command is called periodically to collect stats. As the count is
        // incremented in ISLXML interface file, decrement it here by 1.
        _Module.m_statsLCBroker.m_lCommandsProcessed--;

    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getStats]");
        fSuccess = false;
    }

    return fSuccess;
}
```



```

#include "xc_GetCommands.h"
#include "rs_surgery.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getSurgeryInfo)

bool Cxc_getSurgeryInfo::execCommand()
{
    bool fSuccess = true;

    try
    {
        Crs_surgery rs_surgery;
        string strCpiId;

        CSdoConnection * pconn = m_pcoClient->getConnection();

        //get cpi_id
        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            throw fSuccess = false;
        }

        //check if cpi_id is null.
        if (strCpiId.empty())
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        long lCpiId = atol(strCpiId.c_str());

        //fetch all bp records.
        rs_surgery.clearParms();
        rs_surgery.setRecordSetToNull();
        rs_surgery.setActiveCommand("cmdFetch");
        rs_surgery.setParm("cpi_id", _variant_t (lCpiId));
        if (pconn->execute(rs_surgery) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
        else
        {
            m_pdocResults = new CXmlDocument("<getSurgeryInfo/>");
            fSuccess = rs_surgery.toXml(*m_pdocResults);
        }
    }

    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getSurgeryInfo]");
        fSuccess = false;
    }
}

```



```
    return fSuccess;  
}
```

```
#include "xc_GetCommands.h"
#include "rs_therapy.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getTherapyInfo)

bool Cxc_getTherapyInfo::execCommand()
{
    bool fSuccess = true;

    try
    {
        Crs_therapy rs_therapy;
        string strCpiId;

        CSdoConnection * pconn = m_pcoClient->getConnection();

        //get cpi_id
        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\"cpi_id\" is a required parameter.");
            throw fSuccess = false;
        }

        //check if cpi_id is null.
        if (strCpiId.empty())
        {
            m_emLast.setError("\"cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        long lCpiId = atol(strCpiId.c_str());

        //fetch all bp records.
        rs_therapy.clearParms();
        rs_therapy.setRecordSetToNull();
        rs_therapy.setActiveCommand("cmdFetch");
        rs_therapy.setParm("cpi_id", _variant_t(lCpiId));
        if (pconn->execute(rs_therapy) == false)
        {
            m_emLast.setError(pconn->getLastErrorMessage());
            throw fSuccess = false;
        }
        else
        {
            m_pdocResults = new CXmlDocument("<getTherapyInfo/>");
            fSuccess = rs_therapy.toXml(*m_pdocResults);
        }
    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getTherapyInfo]");
        fSuccess = false;
    }
}
```

```
    return fSuccess;  
}
```

```
#include "xc_getCommands.h"
#include "rs_transfer.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getTransfer)

bool Cxc_getTransfer::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_transfer    rsTransfer;

        string strEncId;
        string strRecId;
        long lEncId = 0;

        //set the command
        rsTransfer.setActiveCommand("cmdFetch");

        //get the parameter
        bool fParamExist = getParm("enc_id", strEncId);
        if (fParamExist) lEncId = atol(strEncId.c_str());

        if (!fParamExist || lEncId == 0)
        {
            m_emLast.setError("\"enc_id\" is a required parameter and should not be 0.");
            return false;
        }

        //set the parameter
        rsTransfer.setParameter("enc_id", _variant_t(lEncId));

        //Do we have the rec_id.....? If yes, then set rec_id parameter.
        if (getParm("rec_id", strRecId) == true)
        {
            long lRecId = atol(strRecId.c_str());
            if (lRecId) rsTransfer.setParameter("rec_id", _variant_t(lRecId));
        }

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsTransfer)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getTransfer/>");
            fSuccess = rsTransfer.toXml(*m_pdocResults);
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getTransfer]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_cpi_master.h"

////////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getUserBiographics)

bool Cxc_getUserBiographics::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_cpi_master rs_cpi_master;
        string strCpiId;

        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\ is a required parameter.");
            throw fSuccess = false;
        }

        //check if cpi_id is null.
        if (strCpiId.empty())
        {
            m_emLast.setError("\cpi_id\ is NULL.");
            throw fSuccess = false;
        }

        long lCpiId = atol(strCpiId.c_str());

        rs_cpi_master.setActiveCommand("cmdFetchUserData");
        rs_cpi_master.setParm("cpi_id", _variant_t (lCpiId));

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rs_cpi_master)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getUserBiographics/>");
            fSuccess = rs_cpi_master.toXml(*m_pdocResults);
        }
    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getUserBiographics]");
        fSuccess = false;
    }
}

return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_insurance.h"
#include "rs_address.h"
#include "rs_phone.h"
```

```
////////////////////////////////////
```

```
CXC_IMPLEMENT_FACTORY(Cxc_getUserInsurance)
```

```
bool Cxc_getUserInsurance::execCommand()
```

```
{
    bool fSuccess = false;

    try
    {
        Crs_phone      rs_phone;
        Crs_address     rs_address;
        Crs_insurance   rsInsurance;

        string strCpiID;
        string strActiveSw;

        rsInsurance.setActiveCommand("cmdFetch");

        if (getParam("cpi_id", strCpiID) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            return false;
        }
        long lCpiId = atol(strCpiID.c_str());
        rsInsurance.setParameter("cpi_id", _variant_t(lCpiId));

        if (getParam("active_sw", strActiveSw) == true)
        {
            _variant_t vActiveSw(atol(strActiveSw.c_str()));
            rsInsurance.setParameter("active_sw", vActiveSw);
        }

        CSdoConnection * pconn = m_pcoClient->getConnection();

        if ((fSuccess = pconn->execute(rsInsurance)) == false)
            m_emLast.setError(pconn->getLastError());

        if (fSuccess)
        {
            m_pdocResults = new CXmlDocument("<getInsuranceInfo/>");

            openXmlTag("insurance", XML_TYPE_GROUP);

            while(!rsInsurance.isBOF() && !rsInsurance.isEOF())
            {
                openXmlTag("row", XML_TYPE_ROW);

                //extract the required fields from recordset and construct result xml
                string strInsuredId = getField(rsInsurance, "cpi_id");
                string strCompanyId = getField(rsInsurance, "ins_company_id");
                string strSubId = getField(rsInsurance, "insured_id");

                //get id's
                //cpi_id is returned from the insured table. so it can be self or
                subscriber id.
                long lSubId = atol(strInsuredId.c_str());
                long lCompanyId = atol(strCompanyId.c_str());

                //create the self_insured_sw
                string strSelfInsuredSw;
```

```

//check if the lSubId is self/subscriber_id
if (lCpiId == lSubId)
{
    //do not send back the subscriber_id as its the cpi_id of the user.
    strSubId = "";
    strSelfInsuredSw = "1";
}
else
{
    strSelfInsuredSw = "0";
}

//populate result xml

//id's
addXmlChild("cpi_id", strCpiID.c_str());
addXmlChild("subscriber_id", strSubId.c_str());
addXmlChild("company_id", strCompanyId.c_str());
addXmlChild("self_insured_sw", strSelfInsuredSw.c_str());
addXmlChild("active_sw", getField(rsInsurance, "active_sw").c_str());
addXmlChild("rec_id", getField(rsInsurance, "rec_id").c_str());

//company info
addXmlChild("name", getField(rsInsurance, "ins_company_name").c_str());
addXmlChild("street1", getField(rsInsurance, "ins_company_street1").c_str() ✓
());
addXmlChild("street2", getField(rsInsurance, "ins_company_street2").c_str() ✓
());
addXmlChild("city", getField(rsInsurance, "ins_company_city").c_str());
addXmlChild("state", getField(rsInsurance, "ins_company_state").c_str());
addXmlChild("state_id", getField(rsInsurance, "ins_company_state_id").c_str() ✓
());
addXmlChild("zip", getField(rsInsurance, "ins_company_zip").c_str());
addXmlChild("country", getField(rsInsurance, "ins_company_country").c_str() ✓
());
addXmlChild("country_id", getField(rsInsurance, "ins_company_country_id").c_str() ✓
());

//insurance info
addXmlChild("plan_id", getField(rsInsurance, "plan_id").c_str());
addXmlChild("plan_code", getField(rsInsurance, "plan_code").c_str());
addXmlChild("plan_type", getField(rsInsurance, "plan_type").c_str());
addXmlChild("effective_dt", getField(rsInsurance, "effective_dt").c_str() ✓
);
addXmlChild("expiration_dt", getField(rsInsurance, "expiration_dt").c_str() ✓
());
addXmlChild("policy_number", getField(rsInsurance, "policy_number").c_str() ✓
());
addXmlChild("group_name", getField(rsInsurance, "group_name").c_str());
addXmlChild("group_number", getField(rsInsurance, "group_number").c_str() ✓
);

//subscriber info (if self insured, then its self info)
addXmlChild("subscriber_last_name", getField(rsInsurance,
"insured_last_name").c_str() ✓
);
addXmlChild("subscriber_first_name", getField(rsInsurance,
"insured_first_name").c_str() ✓
);
addXmlChild("subscriber_middle_name", getField(rsInsurance,
"insured_middle_name").c_str() ✓
);
addXmlChild("subscriber_phone", getField(rsInsurance, "insured_phone").c_str() ✓
);
addXmlChild("subscriber_relationship", getField(rsInsurance,
"insured_relationship").c_str() ✓
);
addXmlChild("subscriber_relationship_id", getField(rsInsurance,
"insured_relationship_id").c_str() ✓
);

```

```

//get the company CLAIMS address
rs_address.clearParms();
rs_address.setRecordSetToNull();
rs_address.setActiveCommand("cmdFetchByPurpose");
rs_address.setParm("cpi_id", _variant_t (lCompanyId));
rs_address.setParm("purpose", _variant_t ("Claims"));
if ((fSuccess = pconn->execute(rs_address)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//populate result xml
if (!rs_address.isEmpty())
{
    addXmlChild("claims_street1", getField(rs_address, "street1").c_str())✓
;
    addXmlChild("claims_street2", getField(rs_address, "street2").c_str())✓
;
    addXmlChild("claims_city", getField(rs_address, "city").c_str());
    addXmlChild("claims_state", getField(rs_address, "state").c_str());
    addXmlChild("claims_state_id", getField(rs_address, "state_id").c_str() ✓
());
    addXmlChild("claims_country", getField(rs_address, "country").c_str())✓
;
    addXmlChild("claims_country_id", getField(rs_address, "country_id"). ✓
c_str());
    addXmlChild("claims_zip", getField(rs_address, "zip").c_str());
}

//get the company phone numbers.
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdFetch");
rs_phone.setParm("cpi_id", _variant_t (lCompanyId));
if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//populate result xml.
while (!rs_phone.isBOF() && !rs_phone.isEOF())
{
    string strTag = "Unknown";
    string strPurpose = getField(rs_phone, "purpose");

    if (strPurpose == "EROOM")        strTag = "phone_emergency";
    if (strPurpose == "MHEALTH")      strTag = "phone_mental_health";
    if (strPurpose == "PRECERT")      strTag = "phone_precert";
    if (strPurpose == "BENEFITS")     strTag = "phone_benefits";
    if (strPurpose == "OTHER")        strTag = "phone_other";

    addXmlChild(strTag, getField(rs_phone, "number").c_str());

    rs_phone.MoveNext();
}

//goto next insurance record
rsInsurance.MoveNext();

//close "row" tag
closeXmlTag();
}

```



```
        //close "insurance" tag
        closeXmlTag();
    }

    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:getUserInsurance]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```

#include "xc_getCommands.h"

#include "rs_hcp.h"
#include "rs_address.h"
#include "rs_name.h"
#include "rs_person.h"

////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getUserPhysicians)

bool Cxc_getUserPhysicians::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_name      rs_name;
        Crs_address    rs_address;
        Crs_person     rs_person;
        Crs_encounter  rs_encounter;
        Crs_hcp_office rs_hcp_office;
        Crs_hcp_specialty rs_hcp_specialty;
        Crs_encounter_hcp rs_encounter_hcp;

        string strCpiId;
        long lCpiId, lHcpId, lRecId, lActiveSw;

        m_pdocResults = NULL;

        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            throw fSuccess = false;
        }

        //check if cpi_id is null.
        if (strCpiId.empty())
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        lCpiId = atol(strCpiId.c_str());

        CSdoConnection * pconn = m_pcoClient->getConnection();
        m_pdocResults = new CXmlDocument("<getUserPhysicians/>");

        /* Get all the user physicians id's */
        rs_encounter_hcp.setActiveCommand("cmdFetchUserPhysicianIds");
        rs_encounter_hcp.setParm("cpi_id", _variant_t (lCpiId));
        if ((fSuccess = pconn->execute(rs_encounter_hcp)) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }

        //create "physicians" tag to encapsulate all physician record
        openXmlTag("physicians", XML_TYPE_GROUP);

        while (!rs_encounter_hcp.isEOF() && !rs_encounter_hcp.isBOF())
        {
            //get hcp id
            string strHcpId, strRecId, strActiveSw;

```

```
rs_encounter_hcp.getField("hcp_id", strHcpId);
rs_encounter_hcp.getField("rec_id", strRecId);
rs_encounter_hcp.getField("active_sw", strActiveSw);
lHcpId = atol(strHcpId.c_str());
lRecId = atol(strRecId.c_str());
lActiveSw = atol(strActiveSw.c_str());
if (!lHcpId || !lRecId) continue;

//create "row" tag to encapsulate each physician record
openXmlTag("row", XML_TYPE_ROW);

//skip these columns from the results and add manually to avoid multiples.
string strColumnsToSkip = "cpi_id,rec_id,active_sw";
addXmlChild("physician_id", lHcpId);
addXmlChild("rec_id", lRecId);
addXmlChild("active_sw", lActiveSw);

//fetch hcp specialty information.
rs_hcp_specialty.setActiveCommand("cmdFetch2");
rs_hcp_specialty.setParm("cpi_id", _variant_t (lHcpId));
if ((fSuccess = pconn->execute(rs_hcp_specialty)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
rs_hcp_specialty.toXml(*m_pdocResults, true, strColumnsToSkip.c_str());

//fetch hcp office information.
//[For now, only one office information will be available]
rs_hcp_office.setActiveCommand("cmdFetchAllInfo");
rs_hcp_office.setParm("cpi_id", _variant_t (lHcpId));
if ((fSuccess = pconn->execute(rs_hcp_office)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
rs_hcp_office.toXml(*m_pdocResults, true, strColumnsToSkip.c_str());

//fetch hcp name
rs_name.setActiveCommand("cmdFetch");
rs_name.setParm("cpi_id", _variant_t (lHcpId));
rs_name.setParm("active_sw", _variant_t ("1"));
if ((fSuccess = pconn->execute(rs_name)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
rs_name.toXml(*m_pdocResults, true, strColumnsToSkip.c_str());

//fetch hcp email address
rs_address.setActiveCommand("cmdFetchEmail");
rs_address.setParm("cpi_id", _variant_t (lHcpId));
rs_address.setParm("purpose", _variant_t ("Email"));
if ((fSuccess = pconn->execute(rs_address)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
rs_address.toXml(*m_pdocResults, true, strColumnsToSkip.c_str());

//fetch hcp personal info
rs_person.setActiveCommand("cmdFetch");
rs_person.setParm("cpi_id", _variant_t (lHcpId));
if ((fSuccess = pconn->execute(rs_person)) == false)
{
    m_emLast.setError(pconn->getLastError());
```

```
        throw fSuccess = false;
    }
    rs_person.toXml(*m_pdocResults, true, strColumnsToSkip.c_str());

    //process next record
    rs_encounter_hcp.MoveNext();

    //close "row" tag
    closeXmlTag();
}

//close "physician" tag
closeXmlTag();

}
catch(bool fError)
{
    fError;
}
catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}
catch(...)
{
    m_emLast.setError("Unknown exception raised. [Command:getUserPhysicians]");
    fSuccess = false;
}

if (!fSuccess && m_pdocResults) delete m_pdocResults;

return fSuccess;
}
```

```
#include "xc_getCommands.h"
#include "rs_user_preference.h"

////////////////////////////////////////

CXC_IMPLEMENT_FACTORY(Cxc_getUserPreference)

bool Cxc_getUserPreference::execCommand()
{
    bool fSuccess = true;

    try
    {
        Crs_user_preference rsUserPreference;
        string strCpiId;

        //get cpi_id
        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\ is a required parameter.");
            throw fSuccess = false;
        }

        //check if cpi_id is null.
        if (strCpiId.empty())
        {
            m_emLast.setError("\cpi_id\ is NULL.");
            throw fSuccess = false;
        }

        long lCpiId = atol(strCpiId.c_str());

        CSdoConnection * pconn = m_pcoClient->getConnection();

        rsUserPreference.setActiveCommand("cmdFetch");
        rsUserPreference.setParm("cpi_id", _variant_t(lCpiId));
        if (pconn->execute(rsUserPreference) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }

        //return the XML result.
        m_pdocResults = new CXmlDocument("<getUserPreference/>");
        fSuccess = rsUserPreference.toXml(*m_pdocResults);

    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised. [Command:getUserPreference]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "xc_GetCommands.h"
#include "rs_blood_pressure.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_getWeightReadings)

bool Cxc_getWeightReadings::execCommand()
{
    bool fSuccess = true;

    try
    {
        Crs_blood_pressure rsBP;
        string strCpiId, strStartDate, strEndDate;

        CSdoConnection * pconn = m_pcoClient->getConnection();

        //get cpi_id
        if (getParam("cpi_id", strCpiId) == false)
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            throw fSuccess = false;
        }

        //optional parameters.
        getParam("start_dt", strStartDate);
        getParam("end_dt", strEndDate);

        //check if cpi_id is null.
        if (strCpiId.empty())
        {
            m_emLast.setError("\cpi_id\" is NULL.");
            throw fSuccess = false;
        }

        long lCpiId = atol(strCpiId.c_str());

        //if start_dt is not provided, then fetch all pulse records.
        if (strStartDate.empty())
        {
            //fetch all pulse records.
            rsBP.setActiveCommand("cmdFetchWeightAll");
            rsBP.setParam("cpi_id", _variant_t (lCpiId));
        }
        else
        {
            COleDateTime oledate;
            DATE dtStartDate;

            //parse start_dt.
            oledate.ParseDateTime(strStartDate.c_str());
            dtStartDate = (DATE) oledate;
            if (dtStartDate == NULL)
            {
                m_emLast.setError("\start_dt\" is Invalid.");
                throw fSuccess = false;
            }

            if (strEndDate.empty())
            {
                //fetch all bp records.
                rsBP.setActiveCommand("cmdFetchWeightByStartDate");
                rsBP.setParam("cpi_id", _variant_t (lCpiId));
                rsBP.setParam("start_dt", _variant_t (dtStartDate));
            }
        }
    }
}
```

```
    else
    {
        DATE dtEndDate;

        //parse end_dt.
        oledate.ParseDateTime(strEndDate.c_str());
        dtEndDate = (DATE) oledate;
        if (dtEndDate == NULL)
        {
            m_emLast.setError("\end_dt\" is Invalid.");
            throw fSuccess = false;
        }

        //get records from start_dt to end_dt
        rsBP.setActiveCommand("cmdFetchWeightByDateRange");
        rsBP.setParm("cpi_id", _variant_t (lCpiId));
        rsBP.setParm("start_dt", _variant_t (dtStartDate));
        rsBP.setParm("end_dt", _variant_t (dtEndDate));
    }

    if (pconn->execute(rsBP) == false)
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }
    else
    {
        m_pdocResults = new CXmlDocument("<getWeightReadings/>");
        fSuccess = rsBP.toXml(*m_pdocResults);
    }

}

catch(bool fError)
{
    fError;
}

catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}

catch(...)
{
    m_emLast.setError("Unkown exception raised. [Command:getWeightReadings]");
    fSuccess = false;
}

return fSuccess;
}
```

```
#include "xc_InsertCommands.h"
```

```
#include "rs_code_cache.h"
```

```
/////////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_insCodeCategory)
```

```
bool Cxc_insCodeCategory::execCommand()  
{
```

```
    //Instantiate the sdo command.  
    Crs_code_cat      rsObject;
```

```
    //set active command  
    rsObject.setActiveCommand("cmdInsert");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rsObject, false);
```

```
    return fSuccess;
```

```
}
```



```
#include "xc_InsertCommands.h"
```

```
#include "rs_cpi_master.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_insCpiMaster)
```

```
bool Cxc_insCpiMaster::execCommand()  
{
```

```
    //Instantiate the sdo command.
```

```
    Crs_cpi_master rsObject;
```

```
    //set active command
```

```
    rsObject.setActiveCommand("cmdInsertEmptyRecord");
```

```
    //update the db.
```

```
    bool fSuccess = executeUpdate(rsObject);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_InsertCommands.h"
```

```
#include "rs_cpi_user.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_insCpiUser)
```

```
bool Cxc_insCpiUser::execCommand()  
{  
  
    //Instantiate the sdo command.  
    Crs_cpi_user    rsObject;  
  
    //set active command  
    rsObject.setActiveCommand("cmdInsert");  
  
    //update the db.  
    bool fSuccess = executeUpdate(rsObject);  
  
    return fSuccess;  
}
```

```
#include "xc_InsertCommands.h"
```

```
#include "rs_diagnosis.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_insDiagnosis)
```

```
bool Cxc_insDiagnosis::execCommand()  
{
```

```
    //Instantiate the sdo command.  
    Crs_diagnosis rsObject;
```

```
    //set active command  
    rsObject.setActiveCommand("cmdInsert");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rsObject);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_InsertCommands.h"
```

```
#include "rs_encounter.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_insEncounter)
```

```
bool Cxc_insEncounter::execCommand()
```

```
{
```

```
    //Instantiate the sdo command.
```

```
    Crs_encounter rsObject;
```

```
    //set active command
```

```
    rsObject.setActiveCommand("cmdInsertEncounter");
```

```
    //update the db.
```

```
    bool fSuccess = executeUpdate(rsObject);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_InsertCommands.h"
```

```
#include "rs_encounter.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_insEncounterLog)
```

```
bool Cxc_insEncounterLog::execCommand()  
{
```

```
    //Instatiate the sdo command.  
    Crs_encounter rsObject;
```

```
    //set active command  
    rsObject.setActiveCommand("cmdInsertEncounterLog");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rsObject);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_InsertCommands.h"
```

```
#include "rs_encounter.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_insEncounterMap)
```

```
bool Cxc_insEncounterMap::execCommand()  
{
```

```
    //Instantiate the sdo command.  
    Crs_encounter_map rsObject;
```

```
    //set active command  
    rsObject.setActiveCommand("cmdInsert");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rsObject);
```

```
    return fSuccess;
```

```
}
```

```
#ifndef xc_insertCommands_h
#define xc_insertCommands_h

#include "stdafx.h"
#include "xcLCBroker.h"

/////////////////////////////////////////////////////////////////
//
// Declaration of all the XML insert Commands Classes.
//
// Macro derives the class from CxcLCBrokerModify
//
/////////////////////////////////////////////////////////////////

DECLARE_XML_INSERTCMD_CLASS(Cxc_insCodeCategory)
DECLARE_XML_INSERTCMD_CLASS(Cxc_insCpiMaster)
DECLARE_XML_INSERTCMD_CLASS(Cxc_insCpiUser)
DECLARE_XML_INSERTCMD_CLASS(Cxc_insDiagnosis)
DECLARE_XML_INSERTCMD_CLASS(Cxc_insEncounterLog)
DECLARE_XML_INSERTCMD_CLASS(Cxc_insEncounterMap)
DECLARE_XML_INSERTCMD_CLASS(Cxc_insExternalCode)
DECLARE_XML_INSERTCMD_CLASS(Cxc_insInternalCode)
DECLARE_XML_INSERTCMD_CLASS(Cxc_insSysOrg)
DECLARE_XML_INSERTCMD_CLASS(Cxc_insEncounter)
DECLARE_XML_INSERTCMD_CLASS(Cxc_insMassMailing)

#endif
```

```
#include "xc_InsertCommands.h"
```

```
#include "rs_code_cache.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_insExternalCode)
```

```
bool Cxc_insExternalCode::execCommand()  
{
```

```
    //Instantiate the sdc command.
```

```
    Crs_code_extern rsObject;
```

```
    //set active command
```

```
    rsObject.setActiveCommand("cmdInsert");
```

```
    //update the db.
```

```
    bool fSuccess = executeUpdate(rsObject);
```

```
    return fSuccess;
```

```
}
```



```
#include "xc_InsertCommands.h"
```

```
#include "rs_code_cache.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_insInternalCode)
```

```
bool Cxc_insInternalCode::execCommand()
```

```
{
```

```
    //Instantiate the sdo command.
```

```
    Crs_code_intern rsObject;
```

```
    //set active command
```

```
    rsObject.setActiveCommand("cmdInsert");
```

```
    //update the db.
```

```
    bool fSuccess = executeUpdate(rsObject, false);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_InsertCommands.h"
```

```
#include "rs_sys_org_facility.h"
```

```
/////////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_insSysOrg)
```

```
bool Cxc_insSysOrg::execCommand()  
{  
    //Instatiate the sdc command.  
    Crs_sys_org    rsObject;  
  
    //set active command  
    rsObject.setActiveCommand("cmdInsert");  
  
    //update the db.  
    bool fSuccess = executeUpdate(rsObject);  
  
    return fSuccess;  
}
```

```
#include "xc_loginUser.h"
#include "Encryptor.h"

#include "rs_cpi_user.h"
#include "rs_name.h"
#include "rs_phone.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_loginUser)

bool Cxc_loginUser::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_cpi_user      rsCpiUser;
        Crs_name           rsName;
        Crs_phone          rsPhone;

        string strUser;
        string strParmPassword;

        if (getParm("user_login", strUser) == false)
        {
            m_emLast.setError("\\"user_login\\" is a required parameter.");
            throw fSuccess = false;
        }
        if (getParm("password", strParmPassword) == false)
        {
            m_emLast.setError("\\"password\\" is a required parameter");
            throw fSuccess = false;
        }

        //get db connection
        CSdoConnection * pconn = m_pcoClient->getConnection();

        //////////////////////////////////////
        // Fetch the user information.
        //////////////////////////////////////

        rsCpiUser.setActiveCommand("cmdFetchUser");
        rsCpiUser.setParameter("user_login", _variant_t(strUser.c_str()));
        if (pconn->execute(rsCpiUser) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }

        if (rsCpiUser.isEmpty())
        {
            m_emLast << "The user [" << strUser << "] does not exist.";
            throw fSuccess = false;
        }

        // password check
        string strEncryptedPassword;
        string strPassword, strPwdReminder;
        string strCpiId;
        string strLastName, strMiddleName, strFirstName;
        string strEffectiveDate, strExpirationDate;
        long lSecurityMask;

        //extract the usefull fields from the result
        rsCpiUser.getField("cpi_id", strCpiId);
        rsCpiUser.getField("password", strEncryptedPassword);
```

```

rsCpiUser.getField("password_reminder", strPwdReminder);
rsCpiUser.getField("effective_dt", strEffectiveDate);
rsCpiUser.getField("expiration_dt", strExpirationDate);
lSecurityMask = (long) rsCpiUser.getField("security_mask");

//check if user is valid.
CEncryptor encryptor;
encryptor.Decrypt(strEncryptedPassword.c_str(), NULL, strPassword);
if (strPassword.compare(strParmPassword) != 0)
{
    m_emLast.setError("Invalid password.");
    throw fSuccess = false;
}

////////////////////////////////////
//User is valid, so fetch some more user data
////////////////////////////////////

DATE dtAccessDate;
dtAccessDate = (DATE) COleDateTime::GetCurrentTime();
long lAuditId = getAuditId();

//update the user access_dt
rsCpiUser.clearParms();
rsCpiUser.setRecordSetToNull();
rsCpiUser.setActiveCommand("cmdUpdate");
rsCpiUser.setParameter("cpi_id", _variant_t(atol(strCpiId.c_str())));
rsCpiUser.setParameter("access_dt", _variant_t(dtAccessDate));
rsCpiUser.setParameter("access_count_sw", _variant_t("1"));
rsCpiUser.setParameter("audit_id", _variant_t(lAuditId));
if ((fSuccess = pconn->execute(rsCpiUser)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//fetch user's name
strLastName = "";
strMiddleName = "";
strFirstName = "";
rsName.clearParms();
rsName.setRecordSetToNull();
rsName.setActiveCommand("cmdFetch");
rsName.setParameter("cpi_id", _variant_t(strCpiId.c_str()));
rsName.setParameter("active_sw", _variant_t("1"));
if (pconn->execute(rsName) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
if (!rsName.isEmpty())
{
    //get user name.
    rsName.getField("last_name", strLastName);
    rsName.getField("middle_name", strMiddleName);
    rsName.getField("first_name", strFirstName);
}

//fetch user's HOME phone number
string strPhone = "";
rsPhone.clearParms();
rsPhone.setRecordSetToNull();
rsPhone.setActiveCommand("cmdFetch");
rsPhone.setParameter("cpi_id", _variant_t(strCpiId.c_str()));
rsPhone.setParameter("active_sw", _variant_t("1"));

```

```

rsPhone.setParameter("purpose", _variant_t("Home"));
rsPhone.setParameter("line_type", _variant_t("Voice"));
if (pconn->execute(rsPhone) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
if (!rsPhone.isEmpty())
{
    //get user phone.
    rsPhone.getField("number", strPhone);
}

////////////////////////////////////
// Construct the result XML
////////////////////////////////////

string strValue;
_variant_t vValue;

m_pdocResults = new CXmlDocument("<loginUser/>");

vValue = strCpiId.c_str();
m_pdocResults->addChild("cpi_id", vValue);
vValue = strEffectiveDate.c_str();
m_pdocResults->addChild("effective_dt", vValue);
vValue = strExpirationDate.c_str();
m_pdocResults->addChild("expiration_dt", vValue);
vValue = strLastName.c_str();
m_pdocResults->addChild("last_name", vValue);
vValue = strMiddleName.c_str();
m_pdocResults->addChild("middle_name", vValue);
vValue = strFirstName.c_str();
m_pdocResults->addChild("first_name", vValue);
vValue = strPhone.c_str();
m_pdocResults->addChild("home_phone", vValue);
vValue = strPwdReminder.c_str();
m_pdocResults->addChild("password_reminder", vValue);

CXmlElement elMasks;
CXmlElement elMask;

m_pdocResults->createElement("security_masks", NOVALUE, &elMasks);
m_pdocResults->addChild(&elMasks);

m_pdocResults->createElement("vip", lSecurityMask & MASK_VIP ? "1" : "0", &elMask) ✓
;
elMasks.addChild(&elMask);
m_pdocResults->createElement("encounter", lSecurityMask & MASK_ENCOUNTER ? "1" :
"0", &elMask);
elMasks.addChild(&elMask);
m_pdocResults->createElement("system", lSecurityMask & MASK_SYSTEM ? "1" : "0", &
elMask);
elMasks.addChild(&elMask);
m_pdocResults->createElement("hcp", lSecurityMask & MASK_HCP ? "1" : "0", &elMask) ✓
;
elMasks.addChild(&elMask);

fSuccess = true;
}
catch(bool fError)
{
    fError;
}
catch(_com_error & e)
{

```

```
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:loginUser]");
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#ifndef xc_openDatabase_h
#define xc_openDatabase_h

class Cxc_openDatabase : public CxcLCBroker
{
public:
    Cxc_openDatabase();
    virtual bool execCommand();
    CXC_DECLARE_FACTORY()
};

#endif
```

```
#ifndef xc_otherCommands_h
#define xc_otherCommands_h
```

```
#include "stdafx.h"
#include "xcLCBroker.h"
```

```
//Specific Commands Include
```

```
#include "xc_openDatabase.h"
#include "xc_loginUser.h"
#include "xc_execSearch.h"
#include "xc_createUser.h"
```

```
////////////////////////////////////
//
// Declaration of the XML Command Classes.
//
// Macro derives the class from CxcLCBroker
//
////////////////////////////////////
```

```
DECLARE_XML_CMD_CLASS(Cxc_changePassword)
```

```
DECLARE_XML_UPDATECMD_CLASS4(Cxc_addInsurance)
```

```
#endif
```



```
#include "xc_OtherCommands.h"
```

```
#include "rs_allergy.h"
```

```
////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_setAllergyInfo)
```

```
////////////////////////////////////
//Do parameter validation here
```

```
bool Cxc_setAllergyInfo::parseParameters ()
{
```

```
    string strData;
```

```
    //////////////////////////////////////
    //cpi_id should be provided.
```

```
    //////////////////////////////////////
    strData = getParameterValue("cpi_id");
```

```
    if (strData.empty())
```

```
    {
        m_emLast.setError("\cpi_id\" is a required parameter.");
        return false;
    }
```

```
    //////////////////////////////////////
    //allergy_name should be provided.
```

```
    //////////////////////////////////////
    strData = getParameterValue("allergy_name");
```

```
    if (strData.empty())
```

```
    {
        m_emLast.setError("\allergy_name\" is a required parameter.");
        return false;
    }
```

```
    //////////////////////////////////////
    //Ensure description always accompanies a code_id
```

```
    //////////////////////////////////////
    if (getParameterValue("allergy_name").empty() && !getParameterValue("allergy_id").
```

```
empty())
```

```
    {
        m_emLast.setError("\allergy_name\" is not present. Codes should be provided with ✓
the code id.");
        return false;
    }
```

```
    if (getParameterValue("type").empty() && !getParameterValue("type_id").empty())
```

```
    {
        m_emLast.setError("\type\" is not present. Codes should be provided with the code ✓
id.");
        return false;
    }
```

```
    if (getParameterValue("severity").empty() && !getParameterValue("severity_id").empty() ✓
())
```

```
    {
        m_emLast.setError("\severity\" is not present. Codes should be provided with the ✓
code id.");
        return false;
    }
```

```
    return true;
```

```
}
```

```

////////////////////////////////////////
//Execute the command.
bool Cxc_setAllergyInfo::execCommand()
{
    //Instantiate the sdc command.
    Crs_allergy rs_allergy;

    //set active command
    rs_allergy.setActiveCommand("cmdUpdate");

    //update the db.
    bool fSuccess = executeUpdate(rs_allergy);

    return fSuccess;
}

/*

////////////////////////////////////////
// Do Data processing here.
// [called from the execute method for each row of data]
//
// - creates insurance company, subscriber and participant insurance records.
//
////////////////////////////////////////
bool Cxc_setAllergyInfo::processData ()
{
    bool fSuccess = true;
    CSdcConnection * pconn = NULL;

    try
    {
        Crs_patient      rs_patient;
        Crs_allergy      rs_allergy;

        string strAllergyName, strType, strSeverity, strReaction, strIdentifyDate;
        long lCpiId, lRecId, lActiveSw, lAllergyId, lTypeId, lSeverityId;

        //get the parameters
        lCpiId = atoi(getParameterValue("cpi_id").c_str());
        lRecId = atoi(getParameterValue("rec_id").c_str());
        lActiveSw = atoi(getParameterValue("active_sw").c_str());
        strAllergyName = getParameterValue("allergy_name");
        lAllergyId = atoi(getParameterValue("Allergy_id").c_str());
        strType = getParameterValue("type");
        lTypeId = atoi(getParameterValue("type_id").c_str());
        strSeverity = getParameterValue("severity");
        lSeverityId = atoi(getParameterValue("severity_id").c_str());
        strIdentifyDate = getParameterValue("identify_dt");
        strReaction = getParameterValue("reaction");

        //convert date in date time object.
        DATE dtIdentifyDate;
        COleDateTime oledate;

        oledate.ParseDateTime(strIdentifyDate.c_str());
        dtIdentifyDate = (DATE) oledate;

        if (!dtIdentifyDate);
    }
}

```

```

    {
        //return error if date is made compulsory.
    }

    //get db connection.
    pconn = m_pcoClient->getConnection();

    //begin transaction
    pconn->beginTrans();

    //get new audit id
    long lAuditId = getAuditId();
    if (!lAuditId)
    {
        m_emLast.setError("Unexpected Condition !!! Cannot get new Audit ID !!!");
        throw fSuccess = false;
    }

    ////////////////////////////////////////////
    //insert/update allergy table
    ////////////////////////////////////////////
    rs_allergy.clearParams();
    rs_allergy.setRecordSetToNull();
    rs_allergy.setActiveCommand("cmdUpdate");
    rs_allergy.setParameter("cpi_id", _variant_t (lCpiId));
    rs_allergy.setParameter("active_sw", _variant_t (lActiveSw));
    rs_allergy.setParameter("allergy_name", _variant_t (strAllergyName.c_str()));
    rs_allergy.setParameter("allergy_id", _variant_t (lAllergyId));
    rs_allergy.setParameter("type", _variant_t (strType.c_str()));
    rs_allergy.setParameter("type_id", _variant_t (lTypeId));
    rs_allergy.setParameter("severity", _variant_t (strSeverity.c_str()));
    rs_allergy.setParameter("severity_id", _variant_t (lSeverityId));
    rs_allergy.setParameter("reaction", _variant_t (strReaction.c_str()));
    rs_allergy.setParameter("audit_id", _variant_t (lAuditId));

    if (dtIdentifyDate) rs_allergy.setParameter("identify_dt", _variant_t
(dtIdentifyDate));

    //updates record if rec_id provided.
    if (lRecId)
        rs_allergy.setParameter("rec_id", _variant_t (lRecId));

    if ((fSuccess = pconn->execute(rs_allergy)) == false)
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }

}
catch(bool fError)
{
    fError;
}
catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}
catch(...)
{
    m_emLast.setError("Unknown exception raised. [Command:setAllergyInfo]");
    fSuccess = false;
}

```

```
//commit or Roll back the transaction.
if (pconn)
{
    if (fSuccess)    pconn->commitTrans();
    else             pconn->rollbackTrans();
}

return fSuccess;
}

*/
```

```
#include "xc_OtherCommands.h"
#include "rs_blood_pressure.h"
```

```
////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_setBloodPressure)
```

```
bool Cxc_setBloodPressure::execCommand()
{
```

```
    bool fSuccess = true;
    CSdoConnection * pconn = NULL;
```

```
    try
    {
```

```
        Crs_blood_pressure      rsBP;
```

```
        string strCpiId, strRecId;
        string strDay, strMonth, strYear, strTime;
        string strSystolicBP, strDiastolicBP, strPulse, strWeight, strWeightUnit;
```

```
        pconn = m_pcoClient->getConnection();
        pconn->beginTransaction();
```

```
        //extract parameters
```

```
        if (getParm("cpi_id", strCpiId) == false)
```

```
        {
            m_emLast.setError("\cpi_id\" is a required parameter.");
            throw fSuccess = false;
        }
```

```
        if (getParm("month", strMonth) == false)
```

```
        {
            m_emLast.setError("\month\" is a required parameter.");
            throw fSuccess = false;
        }
```

```
        if (getParm("day", strDay) == false)
```

```
        {
            m_emLast.setError("\day\" is a required parameter.");
            throw fSuccess = false;
        }
```

```
        if (getParm("year", strYear) == false)
```

```
        {
            m_emLast.setError("\year\" is a required parameter.");
            throw fSuccess = false;
        }
```

```
        if (getParm("time", strTime) == false)
```

```
        {
            m_emLast.setError("\time\" is a required parameter.");
            throw fSuccess = false;
        }
```

```
        if (getParm("systolic_bp", strSystolicBP) == false)
```

```
        {
            m_emLast.setError("\systolic_bp\" is a required parameter.");
            throw fSuccess = false;
        }
```

```
        if (getParm("diastolic_bp", strDiastolicBP) == false)
```

```
        {
            m_emLast.setError("\diastolic_bp\" is a required parameter.");
            throw fSuccess = false;
        }
```

```
        if (getParm("pulse", strPulse) == false)
```

```
        {
            m_emLast.setError("\pulse\" is a required parameter.");
            throw fSuccess = false;
        }
```

```
}
if (getParam("weight", strWeight) == false)
{
    m_emLast.setError("\weight\" is a required parameter");
    throw fSuccess = false;
}

//optional parameters.
getParam("rec_id", strRecId);
getParam("weight_unit", strWeightUnit);

//default values for optional parameters.
if (strWeightUnit.empty()) strWeightUnit = "Pounds";
if (strTime.empty()) strTime = "";

//check if cpi_id is null.
if (strCpiId.empty())
{
    m_emLast.setError("\cpi_id\" is NULL.");
    throw fSuccess = false;
}

//form the date.
COleDateTime oledate;
DATE dt;

string strDate = strMonth + "/" + strDay + "/" + strYear + " " + strTime;
oledate.ParseDateTime(strDate.c_str());
dt = (DATE) oledate;

//check if date is null.
if (dt == NULL)
{
    m_emLast.setError("\Month/Day/Year/Time\" is Invalid or NULL.");
    throw fSuccess = false;
}

//set the command & parameters

rsBP.SetActiveCommand("cmdSetHealthData");
rsBP.setParameter("cpi_id", _variant_t(strCpiId.c_str()));
rsBP.setParameter("systolic_bp", _variant_t(strSystolicBP.c_str()));
rsBP.setParameter("diastolic_bp", _variant_t(strDiastolicBP.c_str()));
rsBP.setParameter("pulse", _variant_t(strPulse.c_str()));
rsBP.setParameter("weight", _variant_t(strWeight.c_str()));
rsBP.setParameter("weight_unit", _variant_t(strWeightUnit.c_str()));
rsBP.setParameter("reading_dt", _variant_t(dt));
rsBP.setParameter("audit_id", _variant_t(getAuditId()));
if (!strRecId.empty())
    rsBP.setParameter("rec_id", _variant_t(strRecId.c_str()));

if (pconn->execute(rsBP) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

}
catch(bool fError)
{
    fError;
}
catch(_com_error & e)
{
    m_emLast.setError(e);
}
```

```
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unkown exception raised. [Command:setBloodPressure]");
        fSuccess = false;
    }

    //commit or Roll back the transaction.
    if (pconn)
    {
        if (fSuccess)    pconn->commitTrans();
        else              pconn->rollbackTrans();
    }

    return fSuccess;
}
```

```
#include "xc_OtherCommands.h"
#include "rs_cholesterol.h"
```

```
////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_setCholesterolReadings)
```

```
bool Cxc_setCholesterolReadings::execCommand()
{
    bool fSuccess = true;

    CSdoConnection * pconn = NULL;

    try
    {
        Crs_cholesterol    rsCholesterol;

        string strCpiId, strRecId;
        string strDay, strMonth, strYear, strTime;
        string strTotalCholesterol, strLdlCholesterol, strHdlCholesterol;

        pconn = m_pcoClient->getConnection();

        //extract parameters
        m_emLast.clear();

        if (getParm("cpi_id", strCpiId) == false)
        {
            m_emLast << "\"cpi_id\" is a required parameter.\r\n";
            fSuccess = false;
        }

        if (getParm("month", strMonth) == false)
        {
            m_emLast << "\"month\" is a required parameter.\n\r";
            fSuccess = false;
        }

        if (getParm("day", strDay) == false)
        {
            m_emLast << "\"day\" is a required parameter.\r\n";
            fSuccess = false;
        }

        if (getParm("year", strYear) == false)
        {
            m_emLast << "\"year\" is a required parameter.\r\n";
            fSuccess = false;
        }

        if (getParm("time", strTime) == false)
        {
            m_emLast << "\"time\" is a required parameter.\r\n";
            fSuccess = false;
        }

        if (getParm("total_cholesterol", strTotalCholesterol) == false)
        {
            m_emLast << "\"cholesterol\" is a required parameter.\r\n";
            fSuccess = false;
        }

        getParm("ldl_cholesterol", strLdlCholesterol);
        getParm("hdl_cholesterol", strHdlCholesterol);

        //optional parameters.
        getParm("rec_id", strRecId);
    }
}
```



```

        //form the date.
        COleDateTime odtReading;
        DATE dateReading;
        string strDate = strMonth + "/" + strDay + "/" + strYear + " " + strTime;
        odtReading.ParseDateTime(strDate.c_str());
        dateReading = (DATE) odtReading;

        //check if date is null.
        if (dateReading == NULL)
        {
            m_emLast << "\"Month/Day/Year/Time\" is Invalid or NULL.\r\n";
            fSuccess = false;
        }

        if (!fSuccess)
            throw fSuccess;

        //set the command & parameters

        rsCholesterol.SetActiveCommand("cmdSetCholesterol");
        rsCholesterol.SetParameter("cpi_id", _variant_t(strCpiId.c_str()));
        if (strRecId.size())
            rsCholesterol.SetParameter("rec_id", _variant_t(strRecId.c_str()));
        rsCholesterol.SetParameter("total_cholesterol", _variant_t(atol
(strTotalCholesterol.c_str())));
        rsCholesterol.SetParameter("ldl_cholesterol", _variant_t(atol(strLdlCholesterol.
c_str())));
        rsCholesterol.SetParameter("hdl_cholesterol", _variant_t(atol(strHdlCholesterol.
c_str())));
        rsCholesterol.SetParameter("reading_dt", _variant_t(dateReading));
        rsCholesterol.SetParameter("audit_id", _variant_t(getAuditId()));

        if (pconn->execute(rsCholesterol) == false)
        {
            m_emLast.SetError(pconn->getLastError());
            throw fSuccess = false;
        }
    }
    catch(bool fError)
    {
        fError;
    }
    catch(_com_error & e)
    {
        m_emLast.SetError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.SetError("Unkown exception raised. [Command:setCholesterol]");
        fSuccess = false;
    }

    return fSuccess;
}

```

```
#ifndef xc_setCommands_h
#define xc_setCommands_h
```

```
#include "stdafx.h"
#include "xcLCBroker.h"
```

```
//Specific Commands Include
```

```
////////////////////////////////////
//
// Declaration of the XML Command Classes.
//
// Macro derives the class from CxcLCBroker
//
////////////////////////////////////
```

```
DECLARE_XML_CMD_CLASS(Cxc_setBloodPressure)
DECLARE_XML_CMD_CLASS(Cxc_setSLMDLocations)
DECLARE_XML_CMD_CLASS(Cxc_setCholesterolReadings)
DECLARE_XML_CMD_CLASS(Cxc_setUnregisteredUser)
```

```
DECLARE_XML_UPDATECMD_CLASS3(Cxc_setUserBiographics)
DECLARE_XML_UPDATECMD_CLASS3(Cxc_setAllergyInfo)
DECLARE_XML_UPDATECMD_CLASS3(Cxc_setHealthConditions)
DECLARE_XML_UPDATECMD_CLASS3(Cxc_setImmunizations)
DECLARE_XML_UPDATECMD_CLASS3(Cxc_setMedications)
DECLARE_XML_UPDATECMD_CLASS3(Cxc_setSurgeryInfo)
DECLARE_XML_UPDATECMD_CLASS3(Cxc_setTherapyInfo)
DECLARE_XML_UPDATECMD_CLASS3(Cxc_setFamilyHistory)
DECLARE_XML_UPDATECMD_CLASS3(Cxc_setImagingInfo)
DECLARE_XML_UPDATECMD_CLASS3(Cxc_setReminder)
DECLARE_XML_UPDATECMD_CLASS3(Cxc_setUserPreference)
```

```
DECLARE_XML_UPDATECMD_CLASS4(Cxc_setEmploymentInfo)
DECLARE_XML_UPDATECMD_CLASS4(Cxc_setUserPhysicians)
DECLARE_XML_UPDATECMD_CLASS4(Cxc_setInsurance)
```

```
#endif
```

```
#include "xc_OtherCommands.h"
```

```
#include "rs_cpi_master.h"
#include "rs_employers.h"
#include "rs_company.h"
#include "rs_address.h"
#include "rs_phone.h"
```

```
////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_setEmploymentInfo)
```

```
////////////////////////////////////////
//Execute the command. [Call execute and write processing code in processData()]
////////////////////////////////////////
bool Cxc_setEmploymentInfo::execCommand()
{
    return execute();
}
```

```
////////////////////////////////////////
//Do parameter validation here
////////////////////////////////////////
bool Cxc_setEmploymentInfo::parseParameters ()
{
```

```
    string  strData;
```

```
    //////////////////////////////////////////
    //cpi_id should be provided.
    //////////////////////////////////////////
    strData = getParameterValue("cpi_id");
    if (strData.empty())
    {
        m_emLast.setError("\cpi_id\" is a required parameter.");
        return false;
    }
```

```
    //////////////////////////////////////////
    //employer name should be provided
    //////////////////////////////////////////
    strData = getParameterValue("name");
    if (strData.empty())
    {
        m_emLast.setError("\name\" is a required.");
        return false;
    }
```

```
    //////////////////////////////////////////
    //Code ID's should be provided if any codes are provided
    //////////////////////////////////////////
    if (!getParameterValue("state").empty() && getParameterValue("state_id").empty())
    {
        m_emLast.setError("\state_id\" is not present. Codes should be accompanied by its
        CodeID.");
        return false;
    }
    if (!getParameterValue("country").empty() && getParameterValue("country_id").empty())
    {
        m_emLast.setError("\country_id\" is not present. Codes should be accompanied by
        its CodeID.");
        return false;
    }
}
```

```

////////////////////////////////////
//Employer rec_id (if provided) should always be accompanied by employer_id
////////////////////////////////////
if (!getParameterValue("rec_id").empty() && getParameterValue("employer_id").empty())
{
    m_emLast.setError("\\"employer_id\\" required if\\"rec_id\\" is provided.");
    return false;
}

return true;
}

////////////////////////////////////
// Do Data processing here.
// [called from the execute method for each row of data]
//
// - creates employer record (in cpi_master) if employer_id is not provided else it updates it. ✓
// - creates employment record (in employment) if rec_id is not provided else it updates it. ✓
// - creates phone and address record if it does not exists else updates it.
//
// * Functions...by parameter provided
//
// - no employer_id & no rec_id : create employer and employment info.
// - employer_id & rec_id : update employer and employment info.
// - employer_id & no rec_id : Employer present, so update employer & create employment info. ✓
// - no employer_id & rec_id : invalid condition.
//
////////////////////////////////////
bool Cxc_setEmploymentInfo::processData ()
{
    bool fSuccess = true;
    CSdoConnection * pconn = NULL;

    try
    {
        Crs_employers rs_employers;
        Crs_company rs_company;
        Crs_cpi_master rs_cpi_master;
        Crs_address rs_address;
        Crs_phone rs_phone;

        string strEmpName, strEmpStreet1, strEmpStreet2, strEmpCity, strEmpState;
        string strEmpZip, strEmpCountry, strEmpPhone, strJobTitle, strEmployeeNumber, ✓
        strStartDate;
        long lCpiId, lEmpCpiId, lEmpRecId, lEmpActiveSw, lEmpStateId, lEmpCountryId;

        //get the data
        lCpiId = atol(getParameterValue("cpi_id").c_str());
        lEmpCpiId = atol(getParameterValue("employer_id").c_str());
        lEmpRecId = atol(getParameterValue("rec_id").c_str()); /* identifies the record ✓
in employment table */
        lEmpActiveSw = atol(getParameterValue("active_sw").c_str());
        strEmpName = getParameterValue("name");
        strEmpStreet1 = getParameterValue("street1");
        strEmpStreet2 = getParameterValue("street2");
        strEmpCity = getParameterValue("city");
        strEmpState = getParameterValue("state");
        lEmpStateId = atol(getParameterValue("state_id").c_str());
        strEmpZip = getParameterValue("zip");
    }
}

```

```

    strEmpCountry = getParameterValue("country");
    lEmpCountryId = atol(getParameterValue("country_id").c_str());
    strEmpPhone = getParameterValue("phone");
    strJobTitle = getParameterValue("job_title");
    strEmployeeNumber = getParameterValue("employee_number");
    strStartDate = getParameterValue("start_dt");

    //if Active Switch not provided, consider default as "1"
    if (!lEmpActiveSw) lEmpActiveSw = 1;

    //convert start_dt to DATE object.
    DATE dtStartDate;
    COleDateTime oledate;
    oledate.ParseDateTime(strStartDate.c_str());
    dtStartDate = (DATE) oledate;

    if (dtStartDate == NULL)
    {
        //return error if start date is made compulsory.
    }

    //get db connection.
    pconn = m_pcoClient->getConnection();

    //begin transaction
    pconn->beginTrans();

    //get new audit id
    long lAuditId = getAuditId();
    if (!lAuditId)
    {
        m_emLast.setError("Unexpected Condition !!! Cannot get new Audit ID !!!");
        throw fSuccess = false;
    }

    //if employer_id is not provided, create new employer record.
    if (!lEmpCpiId)
    {
        //////////////////////////////////////
        //create new company record.
        //////////////////////////////////////

        //get new cpi_id for company
        lEmpCpiId = getNewCpiId();
        if (!lEmpCpiId)
        {
            m_emLast.setError("Unexpected condition!!! Cannot get new Cpi Id for
Employer!!!");
            throw fSuccess = false;
        }

        char szBuffer[20];
        string strCompanyCpiId = "cpi";
        strCompanyCpiId += ltoa(lEmpCpiId, szBuffer, 10);

        //insert new company record in cpi_master
        rs_cpi_master.setActiveCommand("cmdInsertEmptyRecord");
        rs_cpi_master.setParameter("cpi_id", _variant_t (lEmpCpiId));
        rs_cpi_master.setParameter("cpi_text_id", _variant_t (strCompanyCpiId.c_str
    ));
        rs_cpi_master.setParameter("audit_id", _variant_t (lAuditId));
        if ((fSuccess = pconn->execute(rs_cpi_master)) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
    }

```

```

    }

}

////////////////////////////////////////
//update the company record in company table
//(creates a new record if record does not exists)
////////////////////////////////////////

rs_company.setActiveCommand("cmdUpdate");
rs_company.setParameter("cpi_id", _variant_t (lEmpCpiId));
rs_company.setParameter("name", _variant_t (strEmpName.c_str()));
rs_company.setParameter("audit_id", _variant_t (lAuditId));
if ((fSuccess = pconn->execute(rs_company)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

////////////////////////////////////////
//create/update employment record (depends on rec_id)
////////////////////////////////////////

rs_employers.setActiveCommand("cmdUpdate");
rs_employers.setParameter("cpi_id", _variant_t (lCpiId));
rs_employers.setParameter("employer_id", _variant_t (lEmpCpiId));
rs_employers.setParameter("active_sw", _variant_t (lEmpActiveSw));
rs_employers.setParameter("job_title", _variant_t (strJobTitle.c_str()));
rs_employers.setParameter("employee_number", _variant_t (strEmployeeNumber.c_str
(
)));
rs_employers.setParameter("audit_id", _variant_t (lAuditId));

//initilize start_date if available.
if (dtStartDate)
    rs_employers.setParameter("start_dt", _variant_t (dtStartDate));

//set record id (if provided) to update employment record.
if (lEmpRecId)
    rs_employers.setParameter("rec_id", _variant_t (lEmpRecId));

if ((fSuccess = pconn->execute(rs_employers)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

////////////////////////////////////////
//create/update employer phone.
////////////////////////////////////////
string strPhoneRecordId;

//check if employer phone record exists
rs_phone.setActiveCommand("cmdFetchRecordIdByPurpose");
rs_phone.setParameter("cpi_id", _variant_t (lEmpCpiId));
rs_phone.setParameter("purpose", _variant_t ("Work"));
rs_phone.setParameter("line_type", _variant_t ("Voice"));
if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//get the MAX phone rec_id
if (!rs_phone.isEmpty())
    rs_phone.getField("rec_id", strPhoneRecordId);

```

```

rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdUpdate");
rs_phone.setParameter("cpi_id", _variant_t (lEmpCpiId));
rs_phone.setParameter("number", _variant_t (strEmpPhone.c_str()));
rs_phone.setParameter("purpose", _variant_t ("Work"));
rs_phone.setParameter("active_sw", _variant_t ("1"));
rs_phone.setParameter("audit_id", _variant_t (lAuditId));

//set record id if available to update record.
long lRecId = atol(strPhoneRecordId.c_str());
if (lRecId)
    rs_phone.setParameter("rec_id", _variant_t (lRecId));

if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

////////////////////////////////////////
//create/update employer address
////////////////////////////////////////
string strAddressRecordId;

//check if employer address record exists
rs_address.setActiveCommand("cmdFetchRecordIdByPurpose");
rs_address.setParameter("cpi_id", _variant_t (lEmpCpiId));
rs_address.setParameter("purpose", _variant_t ("Work"));
if ((fSuccess = pconn->execute(rs_address)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//get the MAX address rec_id
if (!rs_address.isEmpty())
    rs_address.getField("rec_id", strAddressRecordId);

rs_address.clearParms();
rs_address.setRecordSetToNull();
rs_address.setActiveCommand("cmdUpdate");
rs_address.setParameter("cpi_id", _variant_t (lEmpCpiId));
rs_address.setParameter("active_sw", _variant_t ("1"));
rs_address.setParameter("purpose", _variant_t ("Work"));
rs_address.setParameter("primary_sw", _variant_t ("1"));
rs_address.setParameter("street1", _variant_t (strEmpStreet1.c_str()));
rs_address.setParameter("street2", _variant_t (strEmpStreet2.c_str()));
rs_address.setParameter("city", _variant_t (strEmpCity.c_str()));
rs_address.setParameter("state", _variant_t (strEmpState.c_str()));
if (lEmpStateId) rs_address.setParameter("state_id", _variant_t (lEmpStateId));
rs_address.setParameter("zip", _variant_t (strEmpZip.c_str()));
rs_address.setParameter("country", _variant_t (strEmpCountry.c_str()));
if (lEmpCountryId) rs_address.setParameter("country_id", _variant_t (lEmpCountryId));
rs_address.setParameter("audit_id", _variant_t (lAuditId));

//set record id if available to update record.
lRecId = atol(strAddressRecordId.c_str());
if (lRecId)
    rs_address.setParameter("rec_id", _variant_t (lRecId));

if ((fSuccess = pconn->execute(rs_address)) == false)
{

```

```
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }

}

catch(bool fError)
{
    fError;
}

catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}

catch(...)
{
    m_emLast.setError("Unknown exception raised. [Command:setEmploymentInfo]");
    fSuccess = false;
}

//commit or Roll back the transaction.
if (pconn)
{
    if (fSuccess)    pconn->commitTrans();
    else            pconn->rollbackTrans();
}

return fSuccess;
}
```



```
#include "xc_OtherCommands.h"
```

```
#include "rs_family_history.h"
```

```
////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_setFamilyHistory)
```

```
////////////////////////////////////
//Do parameter validation here
```

```
bool Cxc_setFamilyHistory::parseParameters ()
{
```

```
    string strData;
```

```
    //////////////////////////////////////
    //cpi_id should be provided.
```

```
    //////////////////////////////////////
    strData = getParameterValue("cpi_id");
    if (strData.empty())
```

```
    {
        m_emLast.setError("\cpi_id\" is a required parameter.");
        return false;
    }
```

```
    //////////////////////////////////////
    //Ensure description always accompanies a code_id
```

```
    //////////////////////////////////////
    if (getParameterValue("relationship").empty() && !getParameterValue("relationship_id")✓
        .empty())
```

```
    {
        m_emLast.setError("\relationship\" is not present. Codes should be provided with ✓
        code id.");
        return false;
    }
```

```
    return true;
```

```
}
```

```
////////////////////////////////////
//Execute the command.
```

```
bool Cxc_setFamilyHistory::execCommand()
{
```

```
    //Instantiate the sdc command.
```

```
    Crs_family_history rs_family_history;
```

```
    //set active command
```

```
    rs_family_history.setActiveCommand("cmdUpdate");
```

```
    //update the db.
```

```
    bool fSuccess = executeUpdate(rs_family_history);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_OtherCommands.h"
```

```
#include "rs_health_condition.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_setHealthConditions)
```

```
////////////////////////////////////  
//Do parameter validation here
```

```
bool Cxc_setHealthConditions::parseParameters ()
```

```
{
```

```
    string strData;
```

```
    //////////////////////////////////////  
    //cpi_id should be provided.
```

```
    //////////////////////////////////////  
    strData = getParameterValue("cpi_id");
```

```
    if (strData.empty())
```

```
    {  
        m_emLast.setError("\cpi_id\" is a required parameter.");  
        return false;  
    }
```

```
    //////////////////////////////////////  
    //Ensure description always accompanies a code_id
```

```
    //////////////////////////////////////  
    if (getParameterValue("condition").empty() && !getParameterValue("condition_id").empty()
```

```
    {  
        m_emLast.setError("\condition\" is not present. Codes should be provided with  
code id.");  
        return false;  
    }
```

```
    return true;
```

```
}
```

```
////////////////////////////////////  
//Execute the command.
```

```
bool Cxc_setHealthConditions::execCommand()
```

```
{
```

```
    //Instantiate the sdo command.
```

```
    Crs_health_condition rs_hc;
```

```
    //set active command
```

```
    rs_hc.setActiveCommand("cmdUpdate");
```

```
    //update the db.
```

```
    bool fSuccess = executeUpdate(rs_hc);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_OtherCommands.h"
```

```
#include "rs_imaging.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_setImagingInfo)
```

```
////////////////////////////////////  
//Do parameter validation here
```

```
bool Cxc_setImagingInfo::parseParameters ()  
{
```

```
    string strData;
```

```
    //////////////////////////////////////  
    //cpi_id should be provided.
```

```
    //////////////////////////////////////
```

```
    strData = getParameterValue("cpi_id");
```

```
    if (strData.empty())
```

```
    {  
        m_emLast.setError("\cpi_id\" is a required parameter.");  
        return false;  
    }
```

```
    //////////////////////////////////////  
    //Ensure description always accompanies a code_id
```

```
    //////////////////////////////////////
```

```
    if (getParameterValue("imaging_type").empty() && !getParameterValue("imaging_type_id")  
        .empty())
```

```
    {  
        m_emLast.setError("\imaging_type\" is not present. Codes should be provided with  
code id.");  
        return false;  
    }
```

```
    if (getParameterValue("reason").empty() && !getParameterValue("reason_id").empty())
```

```
    {  
        m_emLast.setError("\reason\" is not present. Codes should be provided with code  
id.");  
        return false;  
    }
```

```
    return true;
```

```
}
```

```
////////////////////////////////////  
//Execute the command.
```

```
bool Cxc_setImagingInfo::execCommand()  
{
```

```
    //Instantiate the sdc command.  
    Crs_imaging rs_imaging;
```

```
    //set active command  
    rs_imaging.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rs_imaging);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_OtherCommands.h"
```

```
#include "rs_immunization.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_setImmunizations)
```

```
////////////////////////////////////  
//Do parameter validation here
```

```
bool Cxc_setImmunizations::parseParameters ()
```

```
{
```

```
    string strData;
```

```
    //////////////////////////////////////  
    //cpi_id should be provided.
```

```
    //////////////////////////////////////
```

```
    strData = getParameterValue("cpi_id");
```

```
    if (strData.empty())
```

```
    {  
        m_emLast.setError("\cpi_id\" is a required parameter.");  
        return false;  
    }
```

```
    //////////////////////////////////////  
    //Ensure description always accompanies a code_id
```

```
    //////////////////////////////////////
```

```
    if (getParameterValue("immunization_type").empty() && !getParameterValue(  
        "immunization_type_id").empty())
```

```
    {  
        m_emLast.setError("\immunization_type\" is not present. Codes should be provided  
        with code id.");  
        return false;  
    }
```

```
    return true;
```

```
}
```

```
////////////////////////////////////  
//Execute the command.
```

```
bool Cxc_setImmunizations::execCommand()
```

```
{
```

```
    //Instantiate the sdo command.
```

```
    Crs_immunization rs_immunization;
```

```
    //set active command
```

```
    rs_immunization.setActiveCommand("cmdUpdate");
```

```
    //update the db.
```

```
    bool fSuccess = executeUpdate(rs_immunization);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_OtherCommands.h"
#include "rs_insurance.h"
```

```
////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_setInsurance)
```

```
////////////////////////////////////
//Execute the command. [Call execute and write processing code in processData()]
////////////////////////////////////
bool Cxc_setInsurance::execCommand()
{
    return execute();
}
```

```
////////////////////////////////////
//Do parameter validation here
////////////////////////////////////
bool Cxc_setInsurance::parseParameters ()
{

```

```
    string strData;
    bool fPresent;
```

```
    //////////////////////////////////////
    //cpi_id should be provided.
    //////////////////////////////////////
    strData = getParameterValue("cpi_id");
    if (strData.empty())
    {
        m_emLast.setError("\cpi_id\" is a required.");
        return false;
    }

```

```
    //////////////////////////////////////
    //rec_id should be provided.
    //////////////////////////////////////
    strData = getParameterValue("rec_id");
    if (strData.empty())
    {
        m_emLast.setError("\rec_id\" is a required.");
        return false;
    }

```

```
    //////////////////////////////////////
    //company_id should be provided.
    //////////////////////////////////////
    strData = getParameterValue("company_id");
    if (strData.empty())
    {
        m_emLast.setError("\company_id\" is a required.");
        return false;
    }

```

```
    //////////////////////////////////////
    //plan_id should be provided. [NO restrictions on plan id for now.]
    //////////////////////////////////////
    /* strData = getParameterValue("plan_id");
    if (strData.empty())
    {
        m_emLast.setError("\plan_id\" is a required.");
        return false;
    }
    */

```

```
*/
```

```

////////////////////////////////////
//Company name should be provided
////////////////////////////////////
strData = getParameterValue("company_name");
if (strData.empty())
{
    m_emLast.setError("\company_name\ is required.");
    return false;
}

////////////////////////////////////
//Self Insured Swith should be provided.
////////////////////////////////////
fPresent = false;
strData = getParameterValue("self_insured_sw");
if (strData.empty())
{
    m_emLast.setError("\self_insured_sw\ is required.");
    return false;
}

bool fSelfInsured = (strData == "1") ? true : false;

////////////////////////////////////
//Subscriber last_name should be provided if any name components are provided
////////////////////////////////////
fPresent = false;
strData = getParameterValue("subscriber_last_name");
if (strData.empty())
{
    if (!getParameterValue("subscriber_first_name").empty()) fPresent = true;
    if (!getParameterValue("subscriber_middle_name").empty()) fPresent = true;

    if (fPresent)
    {
        m_emLast.setError("\Last Name \ is required, if any other name components
are provided.");
        return false;
    }

    //No subscriber info provided, so make sure self_insured switch is "1"
    if (!fSelfInsured)
    {
        m_emLast.setError("No subscriber info provided for dependent !!!");
        return false;
    }
}

////////////////////////////////////
//Code ID's should be provided if any codes are provided
////////////////////////////////////
if (!getParameterValue("state").empty() && getParameterValue("state_id").empty())
{
    m_emLast.setError("\state_id\ is not present. Codes should be accompanied by its
CodeID.");
    return false;
}
if (!getParameterValue("country").empty() && getParameterValue("country_id").empty())
{
    m_emLast.setError("\country_id\ is not present. Codes should be accompanied by
its CodeID.");
    return false;
}
if (!getParameterValue("claims_state").empty() && getParameterValue("claims_state_id").
empty())
{

```

```

        m_emLast.setError("\"claims_state_id\" is not present. Codes should be accompanied
        by its CodeID.");
        return false;
    }
    if (!getParameterValue("claims_country").empty() && getParameterValue(
    "claims_country_id").empty())
    {
        m_emLast.setError("\"claims_country_id\" is not present. Codes should be
        accompanied by its CodeID.");
        return false;
    }
    if (!getParameterValue("subscriber_relationship").empty() && getParameterValue(
    "subscriber_relationship_id").empty())
    {
        m_emLast.setError("\"subscriber_relationship_id\" is not present. Codes should be
        accompanied by its CodeID.");
        return false;
    }

    return true;
}

////////////////////////////////////
// Do Data processing here.
// [called from the execute method for each row of data]
//
// - creates insurance company, subscriber and participant insurance records.
//
////////////////////////////////////
bool Cxc_setInsurance::processData ()
{
    bool fSuccess = true;
    CSdoConnection * pconn = NULL;

    try
    {
        Crs_name          rs_name;
        Crs_address        rs_address;
        Crs_phone          rs_phone;
        Crs_cpi_master     rs_cpi_master;
        Crs_insurance      rs_insurance;
        Crs_insured        rs_insured;
        Crs_company        rs_company;
        Crs_insured_dependents rs_insured_dependents;

        string strCompanyName, strStreet1, strStreet2, strCity, strZip, strState,
        strCountry;
        string strClaimsStreet1, strClaimsStreet2, strClaimsCity, strClaimsState,
        strClaimsZip, strClaimsCountry;
        string strPhoneEmergency, strPhoneMentalHealth, strPhonePreCert, strPhoneBenefits,
        strPhoneOther;
        string strPlanCode, strPlanType, strPlanEffectiveDate, strPolicyNumber,
        strGroupNumber, strGroupName;
        string strSubLastName, strSubFirstName, strSubMiddleName, strSubRelationship,
        strSubPhone;

        long lCpiId, lStateId, lCountryId, lClaimsStateId, lClaimsCountryId,
        lSubRelationshipId;
        bool fSelfInsured, fSubIdPresent;
        long lCompanyCpiId, lInsRecId, lSubCpiId;

        //get the parameters

```

```

lCpiId = atol(getParameterValue("cpi_id").c_str());
lInsRecId = atol(getParameterValue("rec_id").c_str());
lCompanyCpiId = atol(getParameterValue("company_id").c_str());
lSubCpiId = atol(getParameterValue("subscriber_id").c_str());
strCompanyName = getParameterValue("company_name");
strStreet1 = getParameterValue("street1");
strStreet2 = getParameterValue("street2");
strCity = getParameterValue("city");
strState = getParameterValue("state");
lStateId = atol(getParameterValue("state_id").c_str());
strZip = getParameterValue("zip");
strCountry = getParameterValue("country");
lCountryId = atol(getParameterValue("country_id").c_str());
strClaimsStreet1 = getParameterValue("claims_street1");
strClaimsStreet2 = getParameterValue("claims_street2");
strClaimsCity = getParameterValue("claims_city");
strClaimsState = getParameterValue("claims_state");
lClaimsStateId = atol(getParameterValue("claims_state_id").c_str());
strClaimsZip = getParameterValue("claims_zip");
strClaimsCountry = getParameterValue("claims_country");
lClaimsCountryId = atol(getParameterValue("claims_country_id").c_str());
strPhoneEmergency = getParameterValue("phone_emergency");
strPhoneMentalHealth = getParameterValue("phone_mental_health");
strPhonePreCert = getParameterValue("phone_precert");
strPhoneBenefits = getParameterValue("phone_benefits");
strPhoneOther = getParameterValue("phone_other");
strPlanCode = getParameterValue("plan_code");
strPlanType = getParameterValue("plan_type");
strPlanEffectiveDate = getParameterValue("plan_effective_dt");
strPolicyNumber = getParameterValue("policy_number");
strGroupNumber = getParameterValue("group_number");
strGroupName = getParameterValue("group_name");
strSubLastName = getParameterValue("subscriber_last_name");
strSubFirstName = getParameterValue("subscriber_first_name");
strSubMiddleName = getParameterValue("subscriber_middle_name");
strSubRelationship = getParameterValue("subscriber_relationship");
lSubRelationshipId = atol(getParameterValue("subscriber_relationship_id").c_str());

;

strSubPhone = getParameterValue("subscriber_phone");
fSelfInsured = (getParameterValue("self_insured_sw") == "1") ? true : false;

//set flag for subscriber id
fSubIdPresent = (lSubCpiId != 0)? true : false;

//convert date in date time object.
DATE dtPlanEffectiveDate;
COleDateTime oledate;

oledate.ParseDateTime(strPlanEffectiveDate.c_str());
dtPlanEffectiveDate = (DATE)oledate;

if (!dtPlanEffectiveDate)
{
    //return error if date is made compulsory.
}

//get db connection.
pconn = m_pcoClient->getConnection();

//begin transaction
pconn->beginTrans();

//get new audit id
long lAuditId = getAuditId();
if (!lAuditId)
{
    m_emLast.setError("Unexpected Condition !!! Cannot get new Audit ID !!!");
}

```



```

        throw fSuccess = false;
    }

    /*****
    // COMPANY: Update company record
    *****/

    //////////////////////////////////////
    //update company table
    //////////////////////////////////////
    rs_company.clearParms();
    rs_company.setRecordSetToNull();
    rs_company.setActiveCommand("cmdUpdate");
    rs_company.setParameter("cpi_id", _variant_t (lCompanyCpiId));
    rs_company.setParameter("name", _variant_t (strCompanyName.c_str()));
    rs_company.setParameter("audit_id", _variant_t (lAuditId));
    if ((fSuccess = pconn->execute(rs_company)) == false)
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }

    //////////////////////////////////////
    // create/update the company address record.
    //////////////////////////////////////

    string strRecId;
    long lRecId = 0;

    //get rec_id of company address record
    rs_address.clearParms();
    rs_address.setRecordSetToNull();
    rs_address.setActiveCommand("cmdFetchRecordIdByPurpose");
    rs_address.setParameter("cpi_id", _variant_t (lCompanyCpiId));
    rs_address.setParameter("purpose", _variant_t ("Work"));
    if ((fSuccess = pconn->execute(rs_address)) == false)
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }

    //get the MAX rec_id
    if (!rs_address.isEmpty())
    {
        rs_address.getField("rec_id", strRecId);
        lRecId = atol(strRecId.c_str());
    }

    //update company address record
    rs_address.clearParms();
    rs_address.setRecordSetToNull();
    rs_address.setActiveCommand("cmdUpdate");
    rs_address.setParameter("cpi_id", _variant_t (lCompanyCpiId));
    rs_address.setParameter("active_sw", _variant_t ("1"));
    rs_address.setParameter("primary_sw", _variant_t ("1"));
    rs_address.setParameter("purpose", _variant_t ("Work"));
    rs_address.setParameter("street1", _variant_t (strStreet1.c_str()));
    rs_address.setParameter("street2", _variant_t (strStreet2.c_str()));
    rs_address.setParameter("city", _variant_t (strCity.c_str()));
    rs_address.setParameter("state", _variant_t (strState.c_str()));
    rs_address.setParameter("zip", _variant_t (strZip.c_str()));
    rs_address.setParameter("country", _variant_t (strCountry.c_str()));
    rs_address.setParameter("audit_id", _variant_t (lAuditId));

```

```
if (lStateId) rs_address.setParameter("state_id", _variant_t (lStateId));
if (lCountryId) rs_address.setParameter("country_id", _variant_t (lCountryId));

if (lRecId)
    rs_address.setParameter("rec_id", _variant_t (lRecId));

if ((fSuccess = pconn->execute(rs_address)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//get rec_id of company claims address
rs_address.clearParms();
rs_address.setRecordSetToNull();
rs_address.setActiveCommand("cmdFetchRecordIdByPurpose");
rs_address.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_address.setParameter("purpose", _variant_t ("Claims"));
if ((fSuccess = pconn->execute(rs_address)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//get the MAX address rec_id
lRecId = 0;
if (!rs_address.isEmpty())
{
    rs_address.getField("rec_id", strRecId);
    lRecId = atol(strRecId.c_str());
}

//update company claims address record
rs_address.clearParms();
rs_address.setRecordSetToNull();
rs_address.setActiveCommand("cmdUpdate");
rs_address.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_address.setParameter("active_sw", _variant_t ("1"));
rs_address.setParameter("purpose", _variant_t ("Claims"));
rs_address.setParameter("street1", _variant_t (strClaimsStreet1.c_str()));
rs_address.setParameter("street2", _variant_t (strClaimsStreet2.c_str()));
rs_address.setParameter("city", _variant_t (strClaimsCity.c_str()));
rs_address.setParameter("state", _variant_t (strClaimsState.c_str()));
rs_address.setParameter("zip", _variant_t (strClaimsZip.c_str()));
rs_address.setParameter("country", _variant_t (strClaimsCountry.c_str()));
rs_address.setParameter("audit_id", _variant_t (lAuditId));
if (lClaimsStateId) rs_address.setParameter("state_id", _variant_t (lClaimsStateId));
if (lClaimsCountryId) rs_address.setParameter("country_id", _variant_t (lClaimsCountryId));

if (lRecId)
    rs_address.setParameter("rec_id", _variant_t (lRecId));

if ((fSuccess = pconn->execute(rs_address)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//////////
//update company phone records
//////////

//get rec_id for emergency room phone
```

```
lRecId = 0;
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdFetchRecordIdByPurpose");
rs_phone.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_phone.setParameter("purpose", _variant_t ("EROOM"));
rs_phone.setParameter("line_type", _variant_t ("Voice"));
if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
if (!rs_phone.isEmpty())
{
    rs_phone.getField("rec_id", strRecId);
    lRecId = atol(strRecId.c_str());
}

//update emergency room phone
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdUpdate");
rs_phone.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_phone.setParameter("number", _variant_t (strPhoneEmergency.c_str()));
rs_phone.setParameter("purpose", _variant_t ("EROOM"));
rs_phone.setParameter("active_sw", _variant_t ("1"));
rs_phone.setParameter("audit_id", _variant_t (lAuditId));

if (lRecId)
    rs_phone.setParameter("rec_id", _variant_t (lRecId));

if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//get rec_id for mental health phone
lRecId = 0;
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdFetchRecordIdByPurpose");
rs_phone.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_phone.setParameter("purpose", _variant_t ("MHEALTH"));
rs_phone.setParameter("line_type", _variant_t ("Voice"));
if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
if (!rs_phone.isEmpty())
{
    rs_phone.getField("rec_id", strRecId);
    lRecId = atol(strRecId.c_str());
}

//update mental health phone
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdUpdate");
rs_phone.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_phone.setParameter("number", _variant_t (strPhoneMentalHealth.c_str()));
rs_phone.setParameter("purpose", _variant_t ("MHEALTH"));
rs_phone.setParameter("active_sw", _variant_t ("1"));
rs_phone.setParameter("audit_id", _variant_t (lAuditId));

if (lRecId)
```

```
rs_phone.setParameter("rec_id", _variant_t (lRecId));

if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//get rec_id for Pre Certification phone
lRecId = 0;
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdFetchRecordIdByPurpose");
rs_phone.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_phone.setParameter("purpose", _variant_t ("PRECERT"));
rs_phone.setParameter("line_type", _variant_t ("Voice"));
if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
if (!rs_phone.isEmpty())
{
    rs_phone.getField("rec_id", strRecId);
    lRecId = atol(strRecId.c_str());
}

//update Pre Certification phone
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdUpdate");
rs_phone.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_phone.setParameter("number", _variant_t (strPhonePreCert.c_str()));
rs_phone.setParameter("purpose", _variant_t ("PRECERT"));
rs_phone.setParameter("active_sw", _variant_t ("1"));
rs_phone.setParameter("audit_id", _variant_t (lAuditId));

if (lRecId)
    rs_phone.setParameter("rec_id", _variant_t (lRecId));

if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//get rec_id for Benefits phone
lRecId = 0;
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdFetchRecordIdByPurpose");
rs_phone.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_phone.setParameter("purpose", _variant_t ("BENEFITS"));
rs_phone.setParameter("line_type", _variant_t ("Voice"));
if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
if (!rs_phone.isEmpty())
{
    rs_phone.getField("rec_id", strRecId);
    lRecId = atol(strRecId.c_str());
}

//update Benefits phone
rs_phone.clearParms();
```

```

rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdUpdate");
rs_phone.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_phone.setParameter("number", _variant_t (strPhoneBenefits.c_str()));
rs_phone.setParameter("purpose", _variant_t ("BENEFITS"));
rs_phone.setParameter("active_sw", _variant_t ("1"));
rs_phone.setParameter("audit_id", _variant_t (lAuditId));

if (lRecId)
    rs_phone.setParameter("rec_id", _variant_t (lRecId));

if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//get rec_id for Other phone
lRecId = 0;
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdFetchRecordIdByPurpose");
rs_phone.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_phone.setParameter("purpose", _variant_t ("OTHER"));
rs_phone.setParameter("line_type", _variant_t ("Voice"));
if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
if (!rs_phone.isEmpty())
{
    rs_phone.getField("rec_id", strRecId);
    lRecId = atol(strRecId.c_str());
}

//update Other phone
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdUpdate");
rs_phone.setParameter("cpi_id", _variant_t (lCompanyCpiId));
rs_phone.setParameter("number", _variant_t (strPhoneOther.c_str()));
rs_phone.setParameter("purpose", _variant_t ("OTHER"));
rs_phone.setParameter("active_sw", _variant_t ("1"));
rs_phone.setParameter("audit_id", _variant_t (lAuditId));

if (lRecId)
    rs_phone.setParameter("rec_id", _variant_t (lRecId));

if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

/*****
// Update the Subscriber and the Insurance information
*****/

//////////////////////////////////////
///
// Identify the Type of Insurance updation.
// If user is dependant, then update subscriber info.
// If user is self insured, then subscriber info is actually user info, so update
it.

```

```

//////////////////////////////////////✓
///

```

```

long lInsuranceRecId, lInsuredId;
bool fUpdateData = false;

int nSwitchId = 0;
const int nSelfToSelf = 1;
const int nDependentToDependent = 2;
const int nSelfToDependent = 3;
const int nDependentToSelf = 4;

//set proper insurance updation switch
if (fSelfInsured && !fSubIdPresent)      nSwitchId = nSelfToSelf;
if (!fSelfInsured && fSubIdPresent)      nSwitchId = nDependentToDependent;
if (fSelfInsured && fSubIdPresent)      nSwitchId = nDependentToSelf;
if (!fSelfInsured && !fSubIdPresent)    nSwitchId = nSelfToDependent;

```

```

switch(nSwitchId)
{

```

```

case nSelfToSelf:
{
    //subscriber_id not present : previously user was self insured
    //and now user is still self insured

    //update user insured table record.
    lInsuredId = lCpiId;
    lInsuranceRecId = lInsRecId;
    fUpdateData = true;

```

```

    break;
}

```

```

case nDependentToDependent:
{
    //subscriber_id present : previously user was dependent
    //and now user is still dependent.

    //update subscriber insured record and user dependent record
    lInsuredId = lSubCpiId;
    lInsuranceRecId = lInsRecId;
    fUpdateData = true;

```

```

    break;
}

```

```

case nSelfToDependent:
{
    //subscriber_id not present : previously user was self insured
    //but now user is dependent

```

```

    //delete user record from insured table.
    rs_insured.clearParms();
    rs_insured.setRecordSetToNull();
    rs_insured.setActiveCommand("cmdDelete");
    rs_insured.setParameter("cpi_id", _variant_t (lCpiId));
    rs_insured.setParameter("rec_id", _variant_t (lInsRecId));
    if (!pconn->execute(rs_insured))
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }
}

```

```

        //create subscriber in database

        //fetch new cpi_id
        lSubCpiId = getNewCpiId();
        if (!lSubCpiId)
        {
            m_emLast.setError("Unexpected condition!!! Cannot get new cpi id for
subscriber !!!");
            throw fSuccess = false;
        }

        char szBuffer[20];
        string strSubCpiId = "cpi";
        strSubCpiId += ltoa(lSubCpiId, szBuffer, 10);

        //insert new record in cpi_master
        rs_cpi_master.clearParms();
        rs_cpi_master.setRecordSetToNull();
        rs_cpi_master.setActiveCommand("cmdInsertEmptyRecord");
        rs_cpi_master.setParameter("cpi_id", _variant_t (lSubCpiId));
        rs_cpi_master.setParameter("cpi_text_id", _variant_t (strSubCpiId.c_str
()));
        rs_cpi_master.setParameter("audit_id", _variant_t (lAuditId));
        if (!pconn->execute(rs_cpi_master))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }

        //create insurance records.
        lInsuredId = lSubCpiId;
        lInsuranceRecId = 0;
        fUpdateData = false;

        break;
    }

    case nDependentToSelf:
    {
        //subscriber_id present : previously user was dependent
        //but now user is self insured

        //make sure subscriber is not referenced by other users
        rs_insured_dependents.clearParms();
        rs_insured_dependents.setRecordSetToNull();
        rs_insured_dependents.setActiveCommand("cmdCheckReferenceExist");
        rs_insured_dependents.setParameter("insured_cpi_id", _variant_t (lSubCpiId))
;
        rs_insured_dependents.setParameter("dependent_cpi_id", _variant_t
(lCpiId));
        if (!pconn->execute(rs_insured_dependents))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }

        //if subscriber is not referenced then delete the subscriber
        if (rs_insured_dependents.isEmpty())
        {
            //delete subscriber info from cpi_master
            //[it will also deletes insured_dependent table record]
            rs_cpi_master.clearParms();
            rs_cpi_master.setRecordSetToNull();
            rs_cpi_master.setActiveCommand("cmdDelete");

```

```

        rs_cpi_master.setParameter("cpi_id", _variant_t (lSubCpiId));
        if (!pconn->execute(rs_cpi_master))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
    }
    else
    {
        //delete only the insured_dependent record.
        rs_insured_dependents.clearParms();
        rs_insured_dependents.setRecordSetToNull();
        rs_insured_dependents.setActiveCommand("cmdDelete");
        rs_insured_dependents.setParameter("insured_cpi_id", _variant_t
(lSubCpiId));
        rs_insured_dependents.setParameter("dependent_cpi_id", _variant_t
(lCpiId));
        if (!pconn->execute(rs_insured_dependents))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
    }

    //create user insured record
    //pass cpi_id and insured_id as same to create insured table record]
    //lSubCpiId = 0;
    lInsuredId = lCpiId;
    lInsuranceRecId = 0;
    fUpdateData = true;

    break;
}

default:
{
    //execution shouldn't come here.
    lInsuredId = 0;
    lInsuranceRecId = 0;
    fUpdateData = false;
    break;
}

} //end of switch

////////////////////////////////////////
//create/update user/subscriber name & phone records.
////////////////////////////////////////
long lNameRecId = 0;
long lPhoneRecId = 0;

//get the record id's for updating the records.
if (fUpdateData)
{
    //get rec_id for name record
    lNameRecId = 0;
    rs_name.clearParms();
    rs_name.setRecordSetToNull();
    rs_name.setActiveCommand("cmdFetchRecordId");
    rs_name.setParameter("cpi_id", _variant_t (lInsuredId));
    if ((fSuccess = pconn->execute(rs_name)) == false)
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }
    if (!rs_name.isEmpty())

```



```
{
    rs_name.getField("rec_id", strRecId);
    lNameRecId = atol(strRecId.c_str());
    rs_name.setRecordSetToNull();
}

//get rec_id for HOME phone record
lPhoneRecId = 0;
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdFetchRecordIdByPurpose");
rs_phone.setParameter("cpi_id", _variant_t (lInsuredId));
rs_phone.setParameter("purpose", _variant_t ("Home"));
rs_phone.setParameter("line_type", _variant_t ("Voice"));
if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
if (!rs_phone.isEmpty())
{
    rs_phone.getField("rec_id", strRecId);
    lPhoneRecId = atol(strRecId.c_str());
    rs_phone.setRecordSetToNull();
}
}

//update name record
if (!strSubLastName.empty())
{
    rs_name.clearParms();
    rs_name.setRecordSetToNull();
    rs_name.setActiveCommand("cmdUpdate");
    rs_name.setParameter("cpi_id", _variant_t (lInsuredId));
    rs_name.setParameter("active_sw", _variant_t ("1"));
    rs_name.setParameter("last_name", _variant_t (strSubLastName.c_str()));
    rs_name.setParameter("middle_name", _variant_t (strSubMiddleName.c_str()));
    rs_name.setParameter("first_name", _variant_t (strSubFirstName.c_str()));
    rs_name.setParameter("audit_id", _variant_t (lAuditId));

    if (lNameRecId)
        rs_name.setParameter("rec_id", _variant_t (lNameRecId));

    if ((fSuccess = pconn->execute(rs_name)) == false)
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }
}

//update HOME phone record
rs_phone.clearParms();
rs_phone.setRecordSetToNull();
rs_phone.setActiveCommand("cmdUpdate");
rs_phone.setParameter("cpi_id", _variant_t (lInsuredId));
rs_phone.setParameter("number", _variant_t (strSubPhone.c_str()));
rs_phone.setParameter("active_sw", _variant_t ("1"));
rs_phone.setParameter("audit_id", _variant_t (lAuditId));

if (lPhoneRecId)
    rs_phone.setParameter("rec_id", _variant_t (lPhoneRecId));

if ((fSuccess = pconn->execute(rs_phone)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
```

```

    }

    ////////////////////////////////////////////
    //create/update insurance records.
    ////////////////////////////////////////////

    rs_insurance.clearParms();
    rs_insurance.setRecordSetToNull();
    rs_insurance.setActiveCommand("cmdUpdate");
    rs_insurance.setParameter("cpi_id", _variant_t (lCpiId));
    rs_insurance.setParameter("insured_id", _variant_t (lInsuredId));
    if (lInsuranceRecId)
        rs_insurance.setParameter("rec_id", _variant_t (lInsuranceRecId));
    rs_insurance.setParameter("active_sw", _variant_t ("1"));
    rs_insurance.setParameter("ins_co_id", _variant_t (lCompanyCpiId));
    rs_insurance.setParameter("group_name", _variant_t (strGroupName.c_str()));
    rs_insurance.setParameter("group_number", _variant_t (strGroupNumber.c_str()));
    rs_insurance.setParameter("policy_number", _variant_t (strPolicyNumber.c_str()));
    rs_insurance.setParameter("ins_plan_code", _variant_t (strPlanCode.c_str()));
    rs_insurance.setParameter("plan_type_code", _variant_t (strPlanType.c_str()));
    rs_insurance.setParameter("insured_relationship", _variant_t (strSubRelationship.
c_str()));
    rs_insurance.setParameter("audit_id", _variant_t (lAuditId));
    if (dtPlanEffectiveDate)
        rs_insurance.setParameter("plan_eff_dt", _variant_t (dtPlanEffectiveDate));
    if (lSubRelationshipId)
        rs_insurance.setParameter("insured_relationship_id", _variant_t
(lSubRelationshipId));

    if ((fSuccess = pconn->execute(rs_insurance)) == false)
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }

}
catch(bool fError)
{
    fError;
}
catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}
catch(...)
{
    m_emLast.setError("Unknown exception raised. [Command:setInsurance]");
    fSuccess = false;
}

//commit or Roll back the transaction.
if (pconn)
{
    if (fSuccess)    pconn->commitTrans();
    else             pconn->rollbackTrans();
}

return fSuccess;
}

```

```
#include "xc_OtherCommands.h"
```

```
#include "rs_medication.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_setMedications)
```

```
////////////////////////////////////  
//Do parameter validation here
```

```
bool Cxc_setMedications::parseParameters ()  
{
```

```
    string strData;
```

```
    //////////////////////////////////////  
    //cpi_id should be provided.
```

```
    //////////////////////////////////////
```

```
    strData = getParameterValue("cpi_id");
```

```
    if (strData.empty())
```

```
    {  
        m_emLast.setError("\cpi_id\" is a required parameter.");  
        return false;  
    }
```

```
    //////////////////////////////////////  
    //Ensure description always accompanies a code_id
```

```
    //////////////////////////////////////
```

```
    if (getParameterValue("reason").empty() && !getParameterValue("reason_id").empty())
```

```
    {  
        m_emLast.setError("\reason\" is not present. Codes should be provided with code  
id.");  
        return false;
```

```
    }
```

```
    return true;
```

```
}
```

```
////////////////////////////////////  
//Execute the command.
```

```
bool Cxc_setMedications::execCommand()
```

```
{
```

```
    //Instantiate the sdo command.
```

```
    Crs_medication rs_medication;
```

```
    //set active command
```

```
    rs_medication.setActiveCommand("cmdUpdate");
```

```
    //update the db.
```

```
    bool fSuccess = executeUpdate(rs_medication);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_OtherCommands.h"
#include "rs_reminders.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_setReminder)
////////////////////////////////////

//Do parameter validation here
bool Cxc_setReminder::parseParameters ()
{
    bool fSuccess = true;

    string strData;

    // check required parameters
    strData = getParameterValue("cpi_id");
    if (strData.empty())
    {
        m_emLast.setError("\cpi_id\" is a required parameter.");
        fSuccess = false;
    }

    // if doing update then all the following fields are not required
    strData = getParameterValue("rec_id");
    if (!strData.empty() && atol(strData.c_str()) != 0)
        return fSuccess;

    strData = getParameterValue("frequency");
    if (strData.empty())
    {
        m_emLast << "\r\n\"frequency\" is a required parameter.";
        fSuccess = false;
    }

    strData = getParameterValue("time_zone");
    if (strData.empty())
    {
        m_emLast << "\r\n\"time_zone\" is a required parameter.";
        fSuccess = false;
    }

    strData = getParameterValue("time_of_day");
    if (strData.empty())
    {
        m_emLast << "\r\n\"time_of_day\" is a required parameter.";
        fSuccess = false;
    }

    strData = getParameterValue("start_date");
    if (strData.empty())
    {
        m_emLast << "\r\n\"start_date\" is a required parameter.";
        fSuccess = false;
    }

    bool fEmail;
    bool fVoice;
    bool fFax;

    strData = getParameterValue("format_email");
    fEmail = !strData.empty() && strData[0] != '0';
    strData = getParameterValue("format_voice");
    fVoice = !strData.empty() && strData[0] != '0';
    strData = getParameterValue("format_fax");
    fFax = !strData.empty() && strData[0] != '0';

    if (!fEmail && !fVoice && !fFax)
```

```
{
    m_emLast << "\r\nnone of the \"format_*/\" fields must be true.";
    fSuccess = false;
}

strData = getParameterValue("destination");
if (strData.empty())
{
    m_emLast << "\r\n\"destination\" is a required parameter.";
    fSuccess = false;
}

strData = getParameterValue("subject");
if (strData.empty())
{
    m_emLast << "\r\n\"subject\" is a required parameter.";
    fSuccess = false;
}

strData = getParameterValue("body");
if (strData.empty())
{
    m_emLast << "\r\n\"body\" is a required parameter.";
    fSuccess = false;
}

if (!fSuccess)
    return false;

return fSuccess;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Execute the command.
bool Cxc_setReminder::execCommand()
{
    //Instantiate the sdo command.
    Crs_reminder rs_reminder;

    //set active command
    rs_reminder.setActiveCommand("cmdPut");

    //update the db.
    bool fSuccess = executeUpdate(rs_reminder, false);

    return fSuccess;
}
```

```
#include "xc_otherCommands.h"
#include "rs_address.h"
#include "rs_company.h"
#include "rs_cpi_master.h"
```

```
////////////////////////////////////
```

```
CXC_IMPLEMENT_FACTORY(Cxc_setSLMDLocations)
```

```
bool Cxc_setSLMDLocations::execCommand()
{
```

```
    bool fSuccess = true;
```

```
    try
```

```
    {
```

```
        Crs_address rs_address;
        Crs_company rs_company;
        Crs_cpi_master rs_cpi_master;
```

```
        string strName, strStreet1, strStreet2, strCity, strState, strCountry, strZip;
```

```
        //get connection
```

```
        CSdoConnection * pconn = m_pcoClient->getConnection();
```

```
        //extract parameters
```

```
        if (getParam("name", strName) == false)
```

```
        {
```

```
            m_emLast.setError("\name\" is a required parameter.");
            throw fSuccess = false;
```

```
        }
```

```
        if (getParam("street1", strStreet1) == false)
```

```
        {
```

```
            m_emLast.setError("\street1\" is a required parameter.");
            throw fSuccess = false;
```

```
        }
```

```
        if (getParam("zip", strZip) == false)
```

```
        {
```

```
            m_emLast.setError("\zip\" is a required parameter");
            throw fSuccess = false;
```

```
        }
```

```
        //optional parameters.
```

```
        getParam("street2", strStreet2);
```

```
        getParam("city", strCity);
```

```
        getParam("state", strState);
```

```
        getParam("country", strCountry);
```

```
        //check if name is null.
```

```
        if (strName.empty())
```

```
        {
```

```
            m_emLast.setError("Parameter \name\" is NULL.");
            throw fSuccess = false;
```

```
        }
```

```
        //check if street1 is null.
```

```
        if (strStreet1.empty())
```

```
        {
```

```
            m_emLast.setError("Paramter \street1\" is NULL.");
            throw fSuccess = false;
```

```
        }
```

```
        //check if zip is null.
```

```
        if (strZip.empty())
```

```
        {
```

```
            m_emLast.setError("Parameter \zip\" is NULL.");
            throw fSuccess = false;
```

```
)

//get a new cpi_id & audit id's
long lAuditId = getAuditId();
long lCpiId = getNewCpiId();
char szBuffer[20];
string strCpiId = "cpi";
strCpiId += ltoa(lCpiId, szBuffer, 10);
if (lCpiId <= 0)
{
    m_emLast.setError("Error in cpi_id generation.");
    throw fSuccess = false;
}

//write to cpi_master table.
rs_cpi_master.setActiveCommand("cmdInsertEmptyRecord");
rs_cpi_master.setParameter("cpi_id", _variant_t(lCpiId));
rs_cpi_master.setParameter("cpi_text_id", _variant_t(strCpiId.c_str()));
rs_cpi_master.setParameter("audit_id", _variant_t(lAuditId));
if (pconn->execute(rs_cpi_master) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//write to address table.
rs_address.setActiveCommand("cmdUpdate");
rs_address.setParameter("cpi_id", _variant_t(lCpiId));
rs_address.setParameter("street1", _variant_t(strStreet1.c_str()));
rs_address.setParameter("street2", _variant_t(strStreet2.c_str()));
rs_address.setParameter("city", _variant_t(strCity.c_str()));
rs_address.setParameter("state", _variant_t(strState.c_str()));
rs_address.setParameter("country", _variant_t(strCountry.c_str()));
rs_address.setParameter("zip", _variant_t(strZip.c_str()));
rs_address.setParameter("audit_id", _variant_t(lAuditId));
if (pconn->execute(rs_address) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//write to company table.
rs_company.setActiveCommand("cmdUpdate");
rs_company.setParameter("cpi_id", _variant_t(lCpiId));
rs_company.setParameter("name", _variant_t(strName.c_str()));
rs_company.setParameter("audit_id", _variant_t(lAuditId));
if (pconn->execute(rs_company) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}
}
catch(bool fError)
{
    fError;
}
catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}
catch(...)
{
    m_emLast.setError("Unkown exception raised. [Command:setSLMDLocations]");
    fSuccess = false;
}
```

```
    )  
    return fSuccess;  
}
```



```
#include "xc_OtherCommands.h"
```

```
#include "rs_surgery.h"
```

```
////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_setSurgeryInfo)
```

```
////////////////////////////////////
//Do parameter validation here
```

```
bool Cxc_setSurgeryInfo::parseParameters ()
{
```

```
    string strData;
```

```
    //////////////////////////////////////
    //cpi_id should be provided.
```

```
    //////////////////////////////////////
    strData = getParameterValue("cpi_id");
```

```
    if (strData.empty())
```

```
    {
        m_emLast.setError("\cpi_id\" is a required parameter.");
        return false;
    }
```

```
    //////////////////////////////////////
    //Ensure description always accompanies a code_id
```

```
    //////////////////////////////////////
    if (getParameterValue("surgery_type").empty() && !getParameterValue("surgery_type_id").
```

```
empty())
```

```
    {
        m_emLast.setError("\surgery_type\" is not present. Codes should be provided with ✓
code id.");
        return false;
    }
```

```
    return true;
```

```
}
```

```
////////////////////////////////////
//Execute the command.
```

```
bool Cxc_setSurgeryInfo::execCommand()
{
```

```
    //Instantiate the sdo command.
```

```
    Crs_surgery rs_surgery;
```

```
    //set active command
```

```
    rs_surgery.setActiveCommand("cmdUpdate");
```

```
    //update the db.
```

```
    bool fSuccess = executeUpdate(rs_surgery);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_OtherCommands.h"
```

```
#include "rs_therapy.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_setTherapyInfo)
```

```
////////////////////////////////////  
//Do parameter validation here
```

```
bool Cxc_setTherapyInfo::parseParameters ()  
{
```

```
    string strData;
```

```
    //////////////////////////////////////  
    //cpi_id should be provided.
```

```
    //////////////////////////////////////  
    strData = getParameterValue("cpi_id");  
    if (strData.empty())
```

```
    {  
        m_emLast.setError("\cpi_id\" is a required parameter.");  
        return false;  
    }
```

```
    //////////////////////////////////////  
    //Ensure description always accompanies a code_id
```

```
    //////////////////////////////////////  
    if (getParameterValue("therapy_type").empty() && !getParameterValue("therapy_type_id").empty())
```

```
    {  
        m_emLast.setError("\therapy_type\" is not present. Codes should be provided with ✓  
code id.");  
        return false;  
    }
```

```
    if (getParameterValue("reason").empty() && !getParameterValue("reason_id").empty())
```

```
    {  
        m_emLast.setError("\reason\" is not present. Codes should be provided with code ✓  
id.");  
        return false;  
    }
```

```
    return true;
```

```
}
```

```
////////////////////////////////////  
//Execute the command.
```

```
bool Cxc_setTherapyInfo::execCommand()  
{
```

```
    //Instantiate the sdo command.  
    Crs_therapy rs_therapy;
```

```
    //set active command  
    rs_therapy.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rs_therapy);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_OtherCommands.h"

#include "rs_unregistered_user.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_setUnregisteredUser)

bool Cxc_setUnregisteredUser::execCommand()
{
    bool fSuccess = false;

    try
    {
        Crs_unregistered_user rsUnregUser;
        string strUserId;

        //get the user_id.
        if (getParam("user_id", strUserId) == false)
        {
            m_emLast.setError("\\"user_id\\" is a required parameter.");
            throw fSuccess = false;
        }

        //get db connection
        CSdoConnection * pconn = m_pcoClient->getConnection();

        //get current date
        DATE dtCurrentDate;
        dtCurrentDate = (DATE) COleDateTime::GetCurrentTime();
        long lAuditId = getAuditId();
        if (!lAuditId)
        {
            m_emLast.setError("Unexpected error. Could not get new audit id.");
            throw fSuccess = false;
        }

        long lUnregUserId = atol(strUserId.c_str());

        //////////////////////////////////////
        // Check if user id is a valid id.
        //////////////////////////////////////

        rsUnregUser.setActiveCommand("cmdCheckIdExist");
        rsUnregUser.setParameter("user_id", _variant_t(lUnregUserId));
        if (pconn->execute(rsUnregUser) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }

        //////////////////////////////////////
        // Fetch new user id if its invalid.
        //////////////////////////////////////

        if (rsUnregUser.isEmpty())
        {
            //get new id.
            lUnregUserId = getNewUnregUserId();
            if (!lUnregUserId)
            {
                m_emLast.setError("Unexpected error. Could not get new unregistered user id.");
                throw fSuccess = false;
            }
        }

        //update the unreg user record.
        rsUnregUser.clearParms();
    }
}
```

```

        rsUnregUser.setRecordSetToNull();
        rsUnregUser.setActiveCommand("cmdUpdate");
        rsUnregUser.setParameter("user_id", _variant_t(lUnregUserId));
        rsUnregUser.setParameter("effective_dt", _variant_t(dtCurrentDate));
        rsUnregUser.setParameter("access_dt", _variant_t(dtCurrentDate));
        rsUnregUser.setParameter("audit_id", _variant_t(lAuditId));
        if ((fSuccess = pconn->execute(rsUnregUser)) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
    }
    else
    {
        //update the unreg user access_dt
        rsUnregUser.clearParms();
        rsUnregUser.setRecordSetToNull();
        rsUnregUser.setActiveCommand("cmdUpdate");
        rsUnregUser.setParameter("user_id", _variant_t(lUnregUserId));
        rsUnregUser.setParameter("access_dt", _variant_t(dtCurrentDate));
        rsUnregUser.setParameter("access_count_sw", _variant_t("1"));
        rsUnregUser.setParameter("audit_id", _variant_t(lAuditId));
        if ((fSuccess = pconn->execute(rsUnregUser)) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
    }

    ////////////////////////////////////////////
    // construct the XML result.
    // Pass back the existing user_id or the new user_id.
    ////////////////////////////////////////////
    m_pdocResults = new CXmlDocument("<setUnregisteredUser/>");
    m_pdocResults->addChild("user_id", _variant_t(lUnregUserId));

}
catch(bool fError)
{
    fError;
}
catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}
catch(...)
{
    m_emLast.setError("Unkown exception raised. [Command:setUnregisteredUser]");
    fSuccess = false;
}

return fSuccess;
}

```

```
#include "xc_OtherCommands.h"
```

```
#include "rs_cpi_master.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_setUserBiographics)
```

```
////////////////////////////////////  
//Do parameter validation here
```

```
bool Cxc_setUserBiographics::parseParameters ()  
{
```

```
    string strData;  
    bool fPresent;
```

```
    //////////////////////////////////////  
    //cpi_id should be provided.  
    //////////////////////////////////////
```

```
    strData = getParameterValue("cpi_id");  
    if (strData.empty())  
    {  
        m_emLast.setError("\cpi_id\" is a required parameter.");  
        return false;  
    }
```

```
    //////////////////////////////////////  
    //last_name should be provided if any name components are provided  
    //////////////////////////////////////
```

```
    //check if last_name provided  
    fPresent = false;  
    strData = getParameterValue("last_name");  
    if (strData.empty())  
    {  
        //not provided...check if any name components provided  
  
        if (!getParameterValue("first_name").empty()) fPresent = true;  
        if (!getParameterValue("middle_name").empty()) fPresent = true;  
        if (!getParameterValue("nick_name").empty()) fPresent = true;  
        if (!getParameterValue("prefix").empty()) fPresent = true;  
        if (!getParameterValue("suffix").empty()) fPresent = true;  
  
        if (fPresent)  
        {  
            m_emLast.setError("\last_name\" is required, if any other name components are  
provided.");  
            return false;  
        }  
    }
```

```
    //////////////////////////////////////  
    //emergency_last_name should be provided if any emergency name components are provided  
    //////////////////////////////////////
```

```
    //check if last_name provided  
    fPresent = false;  
    strData = getParameterValue("emergency_last_name");  
    if (strData.empty())  
    {  
        //not provided...check if any name components provided  
  
        if (!getParameterValue("emergency_first_name").empty()) fPresent = true;  
        if (!getParameterValue("emergency_middle_name").empty()) fPresent = true;
```

```
        if (fPresent)
        {
            m_emLast.setError("\emergency_last_name\" is required, if any other emergency✓
name components are provided.");
            return false;
        }
    }

    //////////////////////////////////////
    //Code ID's should be provided if any codes are provided
    //////////////////////////////////////
    if (!getParameterValue("state").empty() && getParameterValue("state_id").empty())
    {
        m_emLast.setError("\state_id\" is not present. Codes should be accompanied by its✓
CodeID.");
        return false;
    }
    if (!getParameterValue("gender").empty() && getParameterValue("gender_id").empty())
    {
        m_emLast.setError("\gender_id\" is not present. Codes should be accompanied by ✓
its CodeID.");
        return false;
    }
    if (!getParameterValue("religion").empty() && getParameterValue("religion_id").empty() ✓
    ())
    {
        m_emLast.setError("\religion_id\" is not present. Codes should be accompanied by ✓
its CodeID.");
        return false;
    }
    if (!getParameterValue("marital_status").empty() && getParameterValue(
"marital_status_id").empty())
    {
        m_emLast.setError("\marital_status_id\" is not present. Codes should be ✓
accompanied by its CodeID.");
        return false;
    }
    if (!getParameterValue("emergency_relationship").empty() && getParameterValue(
"emergency_relationship_id").empty())
    {
        m_emLast.setError("\emergency_relationship\" is not present. Codes should be ✓
accompanied by its CodeID.");
        return false;
    }
    if (!getParameterValue("country").empty() && getParameterValue("country_id").empty())
    {
        m_emLast.setError("\country_id\" is not present. Codes should be accompanied by ✓
its CodeID.");
        return false;
    }
    if (!getParameterValue("dl_state").empty() && getParameterValue("dl_state_id").empty() ✓
    ())
    {
        m_emLast.setError("\dl_state_id\" is not present. Codes should be accompanied by ✓
its CodeID.");
        return false;
    }
    if (!getParameterValue("county").empty() && getParameterValue("county_id").empty())
    {
        m_emLast.setError("\county_id\" is not present. Codes should be accompanied by ✓
its CodeID.");
        return false;
    }
    if (!getParameterValue("race").empty() && getParameterValue("race_id").empty())
    {
        m_emLast.setError("\race_id\" is not present. Codes should be accompanied by its ✓
```

```
        CodeID.");
        return false;
    }

    return true;
}

////////////////////////////////////
//Execute the command.
bool Cxc_setUserBiographics::execCommand()
{
    //Instantiate the ado command.
    Crs_cpi_master rs_cpi_master;

    //set active command
    rs_cpi_master.setActiveCommand("cmdSetUserData");

    //update the db.
    bool fSuccess = executeUpdate(rs_cpi_master);

    return fSuccess;
}
```

```

#include "xc_OtherCommands.h"

#include "rs_cpi_master.h"
#include "rs_address.h"
#include "rs_hcp.h"
#include "rs_name.h"
#include "rs_person.h"
#include "rs_encounter.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_setUserPhysicians)

////////////////////////////////////
//Execute the command. [Call execute and write processing code in processData()]
////////////////////////////////////
bool Cxc_setUserPhysicians::execCommand()
{
    return execute();
}

////////////////////////////////////
//Do parameter validation here
////////////////////////////////////
bool Cxc_setUserPhysicians::parseParameters ()
{
    string strData;
    bool fPresent;

    //////////////////////////////////////
    //cpi_id should be provided.
    //////////////////////////////////////
    strData = getParameterValue("cpi_id");
    if (strData.empty())
    {
        m_emLast.setError("\cpi_id\" is a required.");
        return false;
    }

    //////////////////////////////////////
    //Physician last_name should be provided if any name components are provided
    //////////////////////////////////////

    //check if last_name provided
    fPresent = false;
    strData = getParameterValue("last_name");
    if (strData.empty())
    {
        //not provided...check if any name components provided

        if (!getParameterValue("first_name").empty()) fPresent = true;
        if (!getParameterValue("middle_name").empty()) fPresent = true;
        if (fPresent)
        {
            m_emLast.setError("\last_name\" is required, if any other name components are
provided.");
            return false;
        }
    }

    //////////////////////////////////////
    //Code ID's should be provided if any codes are provided
    //////////////////////////////////////
    if (!getParameterValue("office_state").empty() && getParameterValue("office_state_id")✓

```



```

.empty()
{
    m_emLast.setError("\office_state_id\" is not present. Codes should be accompanied✓
    by its CodeID.");
    return false;
}
if (!getParameterValue("office_country").empty() && getParameterValue(
"office_country_id").empty()) ✓
{
    m_emLast.setError("\office_country_id\" is not present. Codes should be ✓
    accompanied by its CodeID.");
    return false;
}
if (!getParameterValue("gender").empty() && getParameterValue("gender_id").empty())
{
    m_emLast.setError("\gender_id\" is not present. Codes should be accompanied by ✓
    its CodeID.");
    return false;
}
if (!getParameterValue("specialty").empty() && getParameterValue("specialty_id").empty()✓
())
{
    m_emLast.setError("\specialty_id\" is not present. Codes should be accompanied by✓
    its CodeID.");
    return false;
}

////////////////////////////////////
//Physician rec_id & physician_id, both should exists if any one is provided.
////////////////////////////////////
if (!getParameterValue("rec_id").empty() && getParameterValue("physician_id").empty())
{
    m_emLast.setError("\physician_id\" required if \"rec_id\" is provided.");
    return false;
}

return true;
}

////////////////////////////////////
// Do Data processing here.
// [called from the execute method for each row of data]
//
// * Functions...by parameters provided
//
// - no physician_id & no rec_id : create physician and hcp link info.
// - physician_id & rec_id : update physician and hcp link info.
// - physician_id & no rec_id : physician present, so update physician & create hcp link ✓
//   info.
// - no physician_id & rec_id : invalid condition.
//
////////////////////////////////////
bool Cxc_setUserPhysicians::processData ()
{
    bool fSuccess = true;
    CSdoConnection * pconn = NULL;

    try
    {
        Crs_name      rs_name;
        Crs_address   rs_address;
        Crs_person    rs_person;
        Crs_encounter rs_encounter;
    }

```

```

    Crs_hcp      rs_hcp;
    Crs_hcp_office rs_hcp_office;
    Crs_cpi_master rs_cpi_master;
    Crs_hcp_specialty rs_hcp_specialty;
    Crs_encounter_hcp rs_encounter_hcp;

    string strHcpSpecialty, strHcpLastName, strHcpFirstName, strHcpMiddleName,
    strHcpGender;
    string strHcpEmailAddress, strOffName, strOffMgrName, strHcpType;
    string strOffZip, strOffCountry, strOffHours, strOffPhone, strOffMgrPhone;
    string strOffStreet1, strOffStreet2, strOffCity, strOffState;

    long lCpiId, lHcpRecId, lHcpCpiId, lHcpActiveSw, lHcpTypeId, lHcpGenderId,
    lHcpSpecialtyId;
    long lOffStateId, lOffCountryId;

    //get the parameters
    lCpiId = atol(getParameterValue("cpi_id").c_str());
    /* identifies the record in encounter_hcp table */
    lHcpRecId = atol(getParameterValue("rec_id").c_str());
    lHcpCpiId = atol(getParameterValue("physician_id").c_str());
    lHcpActiveSw = atol(getParameterValue("active_sw").c_str());
    strHcpLastName = getParameterValue("last_name");
    strHcpFirstName = getParameterValue("first_name");
    strHcpMiddleName = getParameterValue("middle_name");
    strHcpType = getParameterValue("physician_type");
    lHcpTypeId = atol(getParameterValue("physician_type_id").c_str());
    strHcpGender = getParameterValue("gender");
    lHcpGenderId = atol(getParameterValue("gender_id").c_str());
    strHcpSpecialty = getParameterValue("specialty");
    lHcpSpecialtyId = atol(getParameterValue("specialty_id").c_str());
    strHcpEmailAddress = getParameterValue("email_address");
    strOffName = getParameterValue("office_name");
    strOffMgrName = getParameterValue("office_manager");
    strOffMgrPhone = getParameterValue("office_manager_phone");
    strOffStreet1 = getParameterValue("office_street1");
    strOffStreet2 = getParameterValue("office_street2");
    strOffCity = getParameterValue("office_city");
    strOffState = getParameterValue("office_state");
    lOffStateId = atol(getParameterValue("office_state_id").c_str());
    strOffZip = getParameterValue("office_zip");
    strOffCountry = getParameterValue("office_country");
    lOffCountryId = atol(getParameterValue("office_country_id").c_str());
    strOffHours = getParameterValue("office_hours");
    strOffPhone = getParameterValue("office_phone");

    //if Active Switch not provided, consider default as "1"
    if (!lHcpActiveSw)
        lHcpActiveSw = 1;

    //get db connection.
    pconn = m_pcoClient->getConnection();

    //begin transaction
    pconn->beginTrans();

    //get new audit id
    long lAuditId = getAuditId();
    if (!lAuditId)
    {
        m_emLast.setError("Unexpected Condition !!! Cannot get new Audit ID !!!");
        throw fSuccess = false;
    }

    /***
    // ENCOUNTER : Fetch/Create default encounter

```

```

/*****
//check if default encounter present for participant
long lEncId;

rs_encounter.clearParms();
rs_encounter.setRecordSetToNull();
rs_encounter.setActiveCommand("cmdFetchCurrentId");
rs_encounter.setParameter("cpi_id", _variant_t (lCpiId));
if (!pconn->execute(rs_encounter))
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//if default encounter not present, create one.
if (rs_encounter.isEmpty())
{
    //get new encounterId
    lEncId = getNewEncId();
    if (!lEncId)
    {
        m_emLast.setError("Unexpected condition!!! Cannot get new encounter Id!!! ✓
");
        throw fSuccess = false;
    }

    //create default encounter record with the new encounter id
    rs_encounter.clearParms();
    rs_encounter.setRecordSetToNull();
    rs_encounter.setActiveCommand("cmdInsertEncounter");
    rs_encounter.setParameter("enc_id", _variant_t (lEncId));
    rs_encounter.setParameter("cpi_id", _variant_t (lCpiId));
    rs_encounter.setParameter("audit_id", _variant_t (lAuditId));
    if (!pconn->execute(rs_encounter))
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }
}
else
{
    //grab the encounter id.
    string strEncId;
    rs_encounter.getField("enc_id", strEncId);
    lEncId = atol(strEncId.c_str());
    rs_encounter.setRecordSetToNull();
}

//check if we got the encounter id.
if (!lEncId)
{
    m_emLast.setError("Unexpected condition!!! Cannot get new encounter Id!!!");
    throw fSuccess = false;
}

/*****
// HCP : Create/Update Physician information
/*****/

//if hcp id not provided, create new hcp record.
long lRecId = 0;
if (!lHcpCpiId)
{

```

```

////////////////////////////////////
//create new hcp record.
////////////////////////////////////

//get new cpi_id
lHcpCpiId = getNewCpiId();
if (!lHcpCpiId)
{
    m_emLast.setError("Unexpected condition!!! Cannot get new Cpi Id for HCP!!");
    throw fSuccess = false;
}

char szBuffer[20];
string strHcpCpiId = "cpi";
strHcpCpiId += ltoa(lHcpCpiId, szBuffer, 10);

//insert new hcp record in cpi_master
rs_cpi_master.setActiveCommand("cmdInsertEmptyRecord");
rs_cpi_master.setParameter("cpi_id", _variant_t (lHcpCpiId));
rs_cpi_master.setParameter("cpi_text_id", _variant_t (strHcpCpiId.c_str()));
rs_cpi_master.setParameter("audit_id", _variant_t (lAuditId));
if (!pconn->execute(rs_cpi_master))
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

}

////////////////////////////////////
//create/update hcp record
//(creates hcp if hcp not there or updates it)
////////////////////////////////////
rs_hcp.setActiveCommand("cmdUpdate");
rs_hcp.setParameter("cpi_id", _variant_t (lHcpCpiId));
rs_hcp.setParameter("audit_id", _variant_t (lAuditId));
if (!pconn->execute(rs_hcp))
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

////////////////////////////////////
//As only one specialty is currently supported and the update sproc
//inserts multiple specialty records, we have to delete the existing
//specialty records so as to maintain only one record.
////////////////////////////////////
rs_hcp_specialty.setActiveCommand("cmdDeleteAll");
rs_hcp_specialty.setParameter("cpi_id", _variant_t (lHcpCpiId));
if (!pconn->execute(rs_hcp_specialty))
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

////////////////////////////////////
//create/update hcp_specialty info
//[It creates HCP record internally if it doesn't exists]
////////////////////////////////////
rs_hcp_specialty.setActiveCommand("cmdUpdate");
rs_hcp_specialty.setParameter("cpi_id", _variant_t (lHcpCpiId));
if (lHcpSpecialtyId) rs_hcp_specialty.setParameter("specialty_id", _variant_t
(lHcpSpecialtyId));
rs_hcp_specialty.setParameter("specialty", _variant_t (strHcpSpecialty.c_str()));
rs_hcp_specialty.setParameter("audit_id", _variant_t (lAuditId));

```

```

    if (!pconn->execute(rs_hcp_specialty))
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }

    //////////////////////////////////////
    //create/update hcp_office info
    //////////////////////////////////////
    string strHcpOfficeRecId;

    //get record id of hcp_office record to update.
    rs_hcp_office.clearParms();
    rs_hcp_office.setRecordSetToNull();
    rs_hcp_office.setActiveCommand("cmdFetchRecordId");
    rs_hcp_office.setParameter("cpi_id", _variant_t (lHcpCpiId) );
    if (!pconn->execute(rs_hcp_office))
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }
    if (!rs_hcp_office.isEmpty())
    {
        rs_hcp_office.getField("rec_id", strHcpOfficeRecId);
    }

    rs_hcp_office.clearParms();
    rs_hcp_office.setRecordSetToNull();
    rs_hcp_office.setActiveCommand("cmdUpdateAll");
    rs_hcp_office.setParameter("cpi_id", _variant_t (lHcpCpiId) );
    rs_hcp_office.setParameter("name", _variant_t (strOffName.c_str()) );
    rs_hcp_office.setParameter("hours", _variant_t (strOffHours.c_str()) );
    rs_hcp_office.setParameter("mgr_name", _variant_t (strOffMgrName.c_str()) );
    rs_hcp_office.setParameter("mgr_phone", _variant_t (strOffMgrPhone.c_str()) );
    rs_hcp_office.setParameter("phone", _variant_t (strOffPhone.c_str()) );
    rs_hcp_office.setParameter("street1", _variant_t (strOffStreet1.c_str()) );
    rs_hcp_office.setParameter("street2", _variant_t (strOffStreet2.c_str()) );
    rs_hcp_office.setParameter("city", _variant_t (strOffCity.c_str()) );
    rs_hcp_office.setParameter("state", _variant_t (strOffState.c_str()) );
    if (lOffStateId) rs_hcp_office.setParameter("state_id", _variant_t (lOffStateId) );
;
    rs_hcp_office.setParameter("zip", _variant_t (strOffZip.c_str()) );
    rs_hcp_office.setParameter("country", _variant_t (strOffCountry.c_str()) );
    if (lOffCountryId) rs_hcp_office.setParameter("country_id", _variant_t
(lOffCountryId) );
    rs_hcp_office.setParameter("audit_id", _variant_t (lAuditId) );

    //provide rec_id for record updation if it exists.
    lRecId = atol(strHcpOfficeRecId.c_str());
    if (lRecId)
        rs_hcp_office.setParameter("rec_id", _variant_t (lRecId));

    if (!pconn->execute(rs_hcp_office))
    {
        m_emLast.setError(pconn->getLastError());
        throw fSuccess = false;
    }

    //////////////////////////////////////
    //create/update hcp_email address info in address table.
    //////////////////////////////////////
    string strHcpEmailAddressRecId;

    //check if hcp address record exists
    rs_address.clearParms();
    rs_address.setRecordSetToNull();

```

```

rs_address.setActiveCommand("cmdFetchRecordIdByPurpose");
rs_address.setParameter("cpi_id", _variant_t (lHcpCpiId));
rs_address.setParameter("purpose", _variant_t ("Email"));
if ((fSuccess = pconn->execute(rs_address)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//get the MAX address rec_id
if (!rs_address.isEmpty())
    rs_address.getField("rec_id", strHcpEmailAddressRecId);

rs_address.clearParms();
rs_address.setRecordSetToNull();
rs_address.setActiveCommand("cmdUpdate");
rs_address.setParameter("cpi_id", _variant_t (lHcpCpiId));
rs_address.setParameter("active_sw", _variant_t ("1"));
rs_address.setParameter("primary_sw", _variant_t ("0"));
rs_address.setParameter("street1", _variant_t (strHcpEmailAddress.c_str()));
rs_address.setParameter("purpose", _variant_t ("Email"));
rs_address.setParameter("audit_id", _variant_t (lAuditId));

//set record id if available to update record.
lRecId = atol(strHcpEmailAddressRecId.c_str());
if (lRecId)
    rs_address.setParameter("rec_id", _variant_t (lRecId));

if ((fSuccess = pconn->execute(rs_address)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

////////////////////////////////////
//create/update hcp Name info in Name table.
////////////////////////////////////
string strHcpNameRecId;

//check if hcp address record exists
rs_name.clearParms();
rs_name.setRecordSetToNull();
rs_name.setActiveCommand("cmdFetchRecordId");
rs_name.setParameter("cpi_id", _variant_t (lHcpCpiId));
if ((fSuccess = pconn->execute(rs_name)) == false)
{
    m_emLast.setError(pconn->getLastError());
    throw fSuccess = false;
}

//get the MAX name rec_id
if (!rs_name.isEmpty())
    rs_name.getField("rec_id", strHcpNameRecId);

rs_name.clearParms();
rs_name.setRecordSetToNull();
rs_name.setActiveCommand("cmdUpdate");
rs_name.setParameter("cpi_id", _variant_t (lHcpCpiId));
rs_name.setParameter("active_sw", _variant_t ("1"));
rs_name.setParameter("last_name", _variant_t (strHcpLastName.c_str()));
rs_name.setParameter("middle_name", _variant_t (strHcpMiddleName.c_str()));
rs_name.setParameter("first_name", _variant_t (strHcpFirstName.c_str()));
rs_name.setParameter("audit_id", _variant_t (lAuditId));

//set record id if available to update record.
lRecId = atol(strHcpNameRecId.c_str());
if (lRecId)

```

```

        rs_name.setParameter("rec_id", _variant_t (lRecId));

        if ((fSuccess = pconn->execute(rs_name)) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }

        //////////////////////////////////////////
        //create/update hcp Personal info in Person table.
        //////////////////////////////////////////

        rs_person.clearParms();
        rs_person.setRecordSetToNull();
        rs_person.setActiveCommand("cmdUpdate");
        rs_person.setParameter("cpi_id", _variant_t (lHcpCpiId));
        rs_person.setParameter("gender", _variant_t (strHcpGender.c_str()));
        if (lHcpGenderId) rs_person.setParameter("gender_id", _variant_t (lHcpGenderId));
        rs_person.setParameter("audit_id", _variant_t (lAuditId));
        if ((fSuccess = pconn->execute(rs_person)) == false)
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }

        /*-----*/
        // LINK ENCOUNTER_HCP : Link physician to participant
        /*-----*/

        //////////////////////////////////////////
        //  create/update the encounter_hcp table record to link the user & hcp.
        //  (updates if rec_id present, else creates one record)
        //  We have the enc_id and the hcp id.
        //////////////////////////////////////////

        //create encounter_hcp link
        rs_encounter_hcp.clearParms();
        rs_encounter_hcp.setRecordSetToNull();
        rs_encounter_hcp.setActiveCommand("cmdUpdate");
        rs_encounter_hcp.setParameter("enc_id", _variant_t (lEncId ));
        rs_encounter_hcp.setParameter("hcp_id", _variant_t (lHcpCpiId));
        rs_encounter_hcp.setParameter("active_sw", _variant_t (lHcpActiveSw));
        rs_encounter_hcp.setParameter("relation", _variant_t (strHcpType.c_str()));
        if (lHcpTypeId) rs_encounter_hcp.setParameter("relation_id", _variant_t
(lHcpTypeId));
        rs_encounter_hcp.setParameter("audit_id", _variant_t (lAuditId));

        //record id if present, updates record
        if (lHcpRecId)
            rs_encounter_hcp.setParameter("rec_id", _variant_t (lHcpRecId));

        if (!pconn->execute(rs_encounter_hcp))
        {
            m_emLast.setError(pconn->getLastError());
            throw fSuccess = false;
        }
    }

}
catch(bool fError)
{
    fError;
}
catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}

```

```
    }  
    catch(...)   
    {  
        m_emLast.setError("Unknown exception raised. [Command:setUserPhysicians]");  
        fSuccess = false;  
    }  
  
    //commit or Roll back the transaction.  
    if (pconn)  
    {  
        if (fSuccess)    pconn->commitTrans();  
        else              pconn->rollbackTrans();  
    }  
  
    return fSuccess;  
}
```



```
#include "xc_OtherCommands.h"

#include "rs_user_preference.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_setUserPreference)

////////////////////////////////////
//Do parameter validation here
bool Cxc_setUserPreference::parseParameters ()
{
    string strData;

    //////////////////////////////////////
    //cpi_id & preference_id should be provided.
    //////////////////////////////////////
    strData = getParameterValue("cpi_id");
    if (strData.empty())
    {
        m_emLast.setError("\cpi_id\" is a required parameter.");
        return false;
    }

    strData = getParameterValue("preference_id");
    if (strData.empty())
    {
        m_emLast.setError("\preference_id\" is a required parameter.");
        return false;
    }

    return true;
}

////////////////////////////////////
//Execute the command.
bool Cxc_setUserPreference::execCommand()
{
    //Instantiate the sdo command.
    Crs_user_preference rsUserPreference;

    //set active command
    rsUserPreference.setActiveCommand("cmdUpdate");

    //update the db.
    bool fSuccess = executeUpdate(rsUserPreference);

    return fSuccess;
}
```

```
#ifndef xc_updateCommands_h
#define xc_updateCommands_h

#include "stdafx.h"
#include "xcLCBroker.h"

/////////////////////////////////////////////////////////////////
//
// Declaration of all the XML Update Commands Classes.
//
// Macro derives the class from CxcLCBrokerModify to utilize common update routine.
//
/////////////////////////////////////////////////////////////////

DECLARE_XML_UPDATECMD_CLASS(Cxc_uptAddressInfo)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptAdmit)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptCareDirectives)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptCompany)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptConvertPc)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptDiagnosis)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptDischarge)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptEmployment)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptEncounter)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptEncounterHcp)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptEncounterLog)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptExternalCode)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptFacility)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptGuarantor)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptInsurance)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptInsurancePlan)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptLoa)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptPatient)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptPhysical)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptPreAdmit)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptPhone)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptTransfer)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptIdMap)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptNok)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptMiscIds)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptName)
DECLARE_XML_UPDATECMD_CLASS(Cxc_uptPerson)

//declare class with overridden parseCommand() method.
DECLARE_XML_UPDATECMD_CLASS2(Cxc_uptBiographics)

#endif
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_address.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptAddressInfo)
```

```
bool Cxc_uptAddressInfo::execCommand()  
{
```

```
    //Instantiate the sdo command.  
    Crs_address    rsAddress;
```

```
    //set active command  
    rsAddress.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rsAddress);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_admit.h"
```

```
/////////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptAdmit)
```

```
bool Cxc_uptAdmit::execCommand()
```

```
{
```

```
    //Instantiate the sdc command.
```

```
    Crs_admit      rsAdmit;
```

```
    //set active command
```

```
    rsAdmit.setActiveCommand("cmdUpdate");
```

```
    //update the db.
```

```
    bool fSuccess = executeUpdate(rsAdmit);
```

```
    return fSuccess;
```

```
}
```

```

#include "xc_updateCommands.h"

#include "rs_person.h"
#include "rs_name.h"

////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_uptBiographics)
////////////////////////////////////
//Checks for the existance of parameter "data"
bool Cxc_uptBiographics::parseCommand(CXmlDocument * pdoc)
{
    bool fSuccess;
    if (fSuccess = CXmlCommand::parseCommand(pdoc))
        m_docXmlCmd.attach(*pdoc);

    string strDummy;
    fSuccess = (getParm("Name data", strDummy) || getParm("Person data", strDummy));
    if (!fSuccess)
        m_emLast.setError("Missing [Name data]/[Person data] parameters !!!");

    return fSuccess;
}

bool Cxc_uptBiographics::execCommand()
{
    //Instantiate the sdo command.
    Crs_person rs_Person;
    Crs_name rs_Name;

    bool fSuccess = true;
    bool fFound = false;
    string strParm;

    //set active command
    rs_Person.setActiveCommand("cmdUpdate");
    rs_Name.setActiveCommand("cmdUpdate");

    CXMLElement elRoot;
    m_docXmlCmd.getCurrent(&elRoot); //get the root element

    if (getParm("Name data", strParm) == true)
    {
        CXMLElement elNameParm;
        fFound = elRoot.getFirst(&elNameParm); //get the param "Name data" element.
        if (fFound)
        {
            m_docXmlCmd.pushCurrent(&elNameParm); //Push the parm element to the stack. ✓

            fSuccess = execute(rs_Name); //execute
        }
    }

    //execute only if update Name was successfull
    if (fSuccess && (getParm("Person data", strParm) == true))
    {
        CXMLElement elPersonParm;
        if (fFound)
            fFound = elRoot.getNext(&elPersonParm); //get the param "Person data" ✓
        element.
        else
            fFound = elRoot.getFirst(&elPersonParm); //get the param "Person data" ✓
        element.

        if (fFound)
    }

```

```
    {
        m_docXmlCmd.pushCurrent(&elPersonParm); //Push the parm element to the stack. ✓
        fSuccess = execute(rs_Person);           //execute
    }
}

if (!fFound)
{
    m_emLast.setError("No parameters present...[\"Name data\"/\"Person data\"] !!!");
    fSuccess = false;
}

return fSuccess;
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_care_directives.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptCareDirectives)
```

```
bool Cxc_uptCareDirectives::execCommand()  
{  
    //Instatiate the sdo command.  
    Crs_care_directives    rsCareDirectives;  
  
    //set active command.  
    rsCareDirectives.setActiveCommand("cmdUpdate");  
  
    //update the db.  
    bool fSuccess = executeUpdate(rsCareDirectives);  
  
    return fSuccess;  
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_Company.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptCompany)
```

```
bool Cxc_uptCompany::execCommand()  
{
```

```
    //Instatiate the sdc command.  
    Crs_company    rsCompany;
```

```
    //set active command  
    rsCompany.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rsCompany);
```

```
    return fSuccess;
```

```
}
```



```
#include "xc_updateCommands.h"
```

```
#include "rs_convert_pc.h"
```

```
/////////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptConvertPc)
```

```
bool Cxc_uptConvertPc::execCommand()  
{
```

```
    //Instatiate the sdo command.  
    Crs_convert_pc      rsConvertPc;
```

```
    //set active command  
    rsConvertPc.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rsConvertPc);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_diagnosis.h"
```

```
/////////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptDiagnosis)
```

```
bool Cxc_uptDiagnosis::execCommand()  
{  
    //Instantiate the sdc command.  
    Crs_diagnosis      rsDiagnosis;  
  
    //set active command  
    rsDiagnosis.setActiveCommand("cmdUpdate");  
  
    //update the db.  
    bool fSuccess = executeUpdate(rsDiagnosis);  
  
    return fSuccess;  
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_discharge.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptDischarge)
```

```
bool Cxc_uptDischarge::execCommand()  
{
```

```
    //Instantiate the sdc command.  
    Crs_discharge      rsDischarge;
```

```
    //set active command  
    rsDischarge.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rsDischarge);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_employers.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptEmployment)
```

```
bool Cxc_uptEmployment::execCommand()  
{
```

```
    //Instantiate the sdo command.  
    Crs_employers rsEmployment;
```

```
    //set active command  
    rsEmployment.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rsEmployment);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_encounter.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptEncounter)
```

```
bool Cxc_uptEncounter::execCommand()  
{
```

```
    //Updates a record in the Encounter table.
```

```
    //Instantiate the sdo command.
```

```
    Crs_encounter rsEncounter;
```

```
    //set active command
```

```
    rsEncounter.setActiveCommand("cmdUpdateEncounter");
```

```
    //update the db.
```

```
    bool fSuccess = executeUpdate(rsEncounter);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_hcp.h" .. ..
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptEncounterHcp)
```

```
bool Cxc_uptEncounterHcp::execCommand()  
{
```

```
    //Instatiate the sdc command.  
    Crs_encounter_hcp rsEncHcp;
```

```
    //set active command  
    rsEncHcp.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rsEncHcp);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_encounter.h"
```

```
/////////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptEncounterLog)
```

```
bool Cxc_uptEncounterLog::execCommand()  
{
```

```
    //Updates the Encounter. (Updates the Encounter, Discharge, Admit tables).
```

```
    //Instantiate the sdo command.
```

```
    Crs_encounter rsEncounter;
```

```
    //set active command
```

```
    rsEncounter.setActiveCommand("cmdUpdateEncounterLog");
```

```
    //update the db.
```

```
    bool fSuccess = executeUpdate(rsEncounter);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_code_cache.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptExternalCode)
```

```
bool Cxc_uptExternalCode::execCommand()
```

```
{
```

```
    //Instantiate the sdc command.
```

```
    Crs_code_extern rs_code_extern;
```

```
    //set active command
```

```
    rs_code_extern.setActiveCommand("cmdUpdate");
```

```
    //update the db.
```

```
    bool fSuccess = executeUpdate(rs_code_extern);
```

```
    return fSuccess;
```

```
}
```



```
#include "xc_updateCommands.h"
```

```
#include "rs_sys_org_facility.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptFacility)
```

```
bool Cxc_uptFacility::execCommand()  
{
```

```
    //Instantiate the sdo command.  
    Crs_facility rs_facility;
```

```
    //set active command  
    rs_facility.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rs_facility);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_guarantor.h"
```

```
/////////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptGuarantor)
```

```
bool Cxc_uptGuarantor::execCommand()  
{
```

```
    //Instantiate the sdo command.  
    Crs_guarantor rs_guarantor;
```

```
    //set active command  
    rs_guarantor.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rs_guarantor);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_id_map.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptIdMap)
```

```
bool Cxc_uptIdMap::execCommand()  
{
```

```
    //Instatiate the sdc command.  
    Crs_id_map rs_id_map;
```

```
    //set active command  
    rs_id_map.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rs_id_map);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_insurance.h"
```

```
/////////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptInsurance)
```

```
bool Cxc_uptInsurance::execCommand()
```

```
{
```

```
    //Instantiate the sdo command.
```

```
    Crs_insurance rs_insurance;
```

```
    //set active command
```

```
    rs_insurance.setActiveCommand("cmdUpdate");
```

```
    //update the db.
```

```
    bool fSuccess = executeUpdate(rs_insurance);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_insurance.h"
```

```
/////////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptInsurancePlan)
```

```
bool Cxc_uptInsurancePlan::execCommand()  
{
```

```
    //Instantiate the sdc command.
```

```
    Crs_insurance_plan rs_insurance_plan;
```

```
    //set active command
```

```
    rs_insurance_plan.setActiveCommand("cmdUpdate");
```

```
    //update the db.
```

```
    bool fSuccess = executeUpdate(rs_insurance_plan);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_loa.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptLoa)
```

```
bool Cxc_uptLoa::execCommand()
```

```
{
```

```
    //Instantiate the sdc command.
```

```
    Crs_loa rs_loa;
```

```
    //set active command
```

```
    rs_loa.setActiveCommand("cmdUpdate");
```

```
    //update the db.
```

```
    bool fSuccess = executeUpdate(rs_loa);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_misc_id.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptMiscIds)
```

```
bool Cxc_uptMiscIds::execCommand()  
{
```

```
    //Instantiate the sdo command.  
    Crs_misc_id rs_misc_id;
```

```
    //set active command  
    rs_misc_id.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rs_misc_id);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_name.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptName)
```

```
bool Cxc_uptName::execCommand()  
{
```

```
    //Instantiate the sdo command.  
    Crs_name      rsName;
```

```
    //set active command  
    rsName.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rsName);
```

```
    return fSuccess;
```

```
}
```



```
#include "xc_updateCommands.h"
```

```
#include "rs_nok.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptNok)
```

```
bool Cxc_uptNok::execCommand()
```

```
{
```

```
    //Instantiate the sdo command.  
    Crs_nok rs_nok;
```

```
    //set active command  
    rs_nok.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rs_nok);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_patient.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptPatient)
```

```
bool Cxc_uptPatient::execCommand()  
{
```

```
    //Instantiate the sdo command.  
    Crs_patient rs_patient;
```

```
    //set active command  
    rs_patient.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rs_patient);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_person.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptPerson)
```

```
bool Cxc_uptPerson::execCommand()  
{
```

```
    //Instatiate the sdc command.  
    Crs_person    rsPerson;
```

```
    //set active command  
    rsPerson.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rsPerson);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_phone.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptPhone)
```

```
bool Cxc_uptPhone::execCommand()  
{
```

```
    //Instantiate the sdc command.  
    Crs_phone rs_phone;
```

```
    //set active command  
    rs_phone.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rs_phone);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_physical.h"
```

```
////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptPhysical)
```

```
bool Cxc_uptPhysical::execCommand()  
{
```

```
    //instantiate the sdo command.  
    Crs_physical rs_physical;
```

```
    //set active command  
    rs_physical.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rs_physical);
```

```
    return fSuccess;
```

```
}
```

```
#include "xc_updateCommands.h",
#include "rs_pre_admit.h"

////////////////////////////////////////
CXC_IMPLEMENT_FACTORY(Cxc_uptPreAdmit)

bool Cxc_uptPreAdmit::execCommand()
{
    //Instantiate the sdo command.
    Crs_pre_admit rs_pre_admit;

    //set active command
    rs_pre_admit.setActiveCommand("cmdUpdate");

    //update the db.
    bool fSuccess = executeUpdate(rs_pre_admit);

    return fSuccess;
}
```

```
#include "xc_updateCommands.h"
```

```
#include "rs_transfer.h"
```

```
/////////////////////////////////////////  
CXC_IMPLEMENT_FACTORY(Cxc_uptTransfer)
```

```
bool Cxc_uptTransfer::execCommand()  
{
```

```
    //Instantiate the sdo command.  
    Crs_transfer rs_transfer;
```

```
    //set active command  
    rs_transfer.setActiveCommand("cmdUpdate");
```

```
    //update the db.  
    bool fSuccess = executeUpdate(rs_transfer);
```

```
    return fSuccess;
```

```
}
```

```
////////////////////////////////////
// Include files
////////////////////////////////////

#include "stdafx.h"
#include "SldoBase.h"
#include "idGen.h"
#include "xcLCBroker.h"

//Get Commands Include
#include "xc_GetCommands.h"

//Set Commands Include
#include "xc_SetCommands.h"

//Update command includes
#include "xc_UpdateCommands.h"

//delete command includes
#include "xc_deleteCommands.h"

//Insert command includes
#include "xc_InsertCommands.h"

//Other command includes
#include "xc_OtherCommands.h"

////////////////////////////////////
// CxcLCBrokerFactory
////////////////////////////////////

CXC_FACTORY_MAP(CxcLCBrokerFactory)
/*!!!!!!!!!!!!!! keep in alphabetical order, searched binary !!!!!!!!!!!!!!!*/
//delete commands

CXC_FACTORY_ENTRY(addInsurance)
CXC_FACTORY_ENTRY(changePassword)
CXC_FACTORY_ENTRY(createUser)
CXC_FACTORY_ENTRY(delAudit)
CXC_FACTORY_ENTRY(delDiagnosis)
CXC_FACTORY_ENTRY(deleteAllergy)
CXC_FACTORY_ENTRY(deleteEmploymentInfo)
CXC_FACTORY_ENTRY(deleteFamilyHistory)
CXC_FACTORY_ENTRY(deleteHealthConditions)
CXC_FACTORY_ENTRY(deleteImagingInfo)
CXC_FACTORY_ENTRY(deleteImmunizations)
CXC_FACTORY_ENTRY(deleteInsurance)
CXC_FACTORY_ENTRY(deleteMedications)
CXC_FACTORY_ENTRY(deletePhysical)
CXC_FACTORY_ENTRY(deleteReminder)
CXC_FACTORY_ENTRY(deleteSurgeryInfo)
CXC_FACTORY_ENTRY(deleteTherapyInfo)
CXC_FACTORY_ENTRY(deleteUnregisteredUser)
CXC_FACTORY_ENTRY(deleteUserPhysician)
CXC_FACTORY_ENTRY(delHcpOffice)
CXC_FACTORY_ENTRY(delHcpSpecialty)

//search commands
CXC_FACTORY_ENTRY(execSearch)

//get commands
CXC_FACTORY_ENTRY(getAccountInfo)
CXC_FACTORY_ENTRY(getAddressInfo)
```



```
CXC_FACTORY_ENTRY(getAdmit)
CXC_FACTORY_ENTRY(getAllergyInfo)
CXC_FACTORY_ENTRY(getBeds)
CXC_FACTORY_ENTRY(getBiographicalInfo)
CXC_FACTORY_ENTRY(getBloodPressureReadings)
CXC_FACTORY_ENTRY(getCareDirectives)
CXC_FACTORY_ENTRY(getCholesterolReadings)
CXC_FACTORY_ENTRY(getCodeCats)
CXC_FACTORY_ENTRY(getCodes)
CXC_FACTORY_ENTRY(getCompany)
CXC_FACTORY_ENTRY(getConvertPc)
CXC_FACTORY_ENTRY(getCpiId)
CXC_FACTORY_ENTRY(getCpiIdExists)
CXC_FACTORY_ENTRY(getCurrConvertPc)
CXC_FACTORY_ENTRY(getCurrEncounter)
CXC_FACTORY_ENTRY(getCurrEncounterId)
CXC_FACTORY_ENTRY(getCurrLoa)
CXC_FACTORY_ENTRY(getCurrPreAdmit)
CXC_FACTORY_ENTRY(getCurrTransfer)
CXC_FACTORY_ENTRY(getDiagnosis)
CXC_FACTORY_ENTRY(getDisability)
CXC_FACTORY_ENTRY(getDischarge)
CXC_FACTORY_ENTRY(getDischargeHistory)
CXC_FACTORY_ENTRY(getEmploymentInfo)
CXC_FACTORY_ENTRY(getEncounterTree)
CXC_FACTORY_ENTRY(getExternalIDs)
CXC_FACTORY_ENTRY(getFacilities)
CXC_FACTORY_ENTRY(getFamilyHistory)
CXC_FACTORY_ENTRY(getFamilyTree)
CXC_FACTORY_ENTRY(getGuarantorInfo)
CXC_FACTORY_ENTRY(getHealthConditions)
CXC_FACTORY_ENTRY(getIdealBPRanges)
CXC_FACTORY_ENTRY(getImagingInfo)
CXC_FACTORY_ENTRY(getImmunizations)
CXC_FACTORY_ENTRY(getInPatients)
CXC_FACTORY_ENTRY(getInsPlans)
CXC_FACTORY_ENTRY(getInsPlansByCompany)
CXC_FACTORY_ENTRY(getInsuranceCoverage)
CXC_FACTORY_ENTRY(getInsuranceInfo)
CXC_FACTORY_ENTRY(getLifeclinicStats)
CXC_FACTORY_ENTRY(getLoa)
CXC_FACTORY_ENTRY(getLoaHistory)
CXC_FACTORY_ENTRY(getMassMailing)
CXC_FACTORY_ENTRY(getMedications)
CXC_FACTORY_ENTRY(getMiscIDs)
CXC_FACTORY_ENTRY(getName)
CXC_FACTORY_ENTRY(getNewEncounterId)
CXC_FACTORY_ENTRY(getNewUnregUserId)
CXC_FACTORY_ENTRY(getNok)
CXC_FACTORY_ENTRY(getNokAll)
CXC_FACTORY_ENTRY(getPasswordReminder)
CXC_FACTORY_ENTRY(getPatientLocation)
CXC_FACTORY_ENTRY(getPatientStatus)
CXC_FACTORY_ENTRY(getPatientValuables)
CXC_FACTORY_ENTRY(getPerson)
CXC_FACTORY_ENTRY(getPhone)
CXC_FACTORY_ENTRY(getPhysicalInfo)
CXC_FACTORY_ENTRY(getPhysicianInfo)
CXC_FACTORY_ENTRY(getPhysicians)
CXC_FACTORY_ENTRY(getPocs)
CXC_FACTORY_ENTRY(getPreAdmit)
CXC_FACTORY_ENTRY(getPulseReadings)
CXC_FACTORY_ENTRY(getReminder)
CXC_FACTORY_ENTRY(getRooms)
CXC_FACTORY_ENTRY(getSecurityInfo)
CXC_FACTORY_ENTRY(getSLMDLocations)
CXC_FACTORY_ENTRY(getStats)
```

```
CXC_FACTORY_ENTRY(getSurgeryInfo)
CXC_FACTORY_ENTRY(getTherapyInfo)
CXC_FACTORY_ENTRY(getTransfer)
CXC_FACTORY_ENTRY(getUserBiographics)
CXC_FACTORY_ENTRY(getUserInsurance)
CXC_FACTORY_ENTRY(getUserPhysicians)
CXC_FACTORY_ENTRY(getUserPreference)
CXC_FACTORY_ENTRY(getWeightReadings)

//insert commands
CXC_FACTORY_ENTRY(insCodeCategory)
CXC_FACTORY_ENTRY(insCpiMaster)
CXC_FACTORY_ENTRY(insCpiUser)
CXC_FACTORY_ENTRY(insDiagnosis)
CXC_FACTORY_ENTRY(insEncounter)
CXC_FACTORY_ENTRY(insEncounterLog)
CXC_FACTORY_ENTRY(insEncounterMap)
CXC_FACTORY_ENTRY(insExternalCode)
CXC_FACTORY_ENTRY(insInternalCode)
CXC_FACTORY_ENTRY(insMassMailing)
CXC_FACTORY_ENTRY(insSysOrg)

//general commands
CXC_FACTORY_ENTRY(loginUser)
CXC_FACTORY_ENTRY(openDatabase)
CXC_FACTORY_ENTRY(setAllergyInfo)
CXC_FACTORY_ENTRY(setBloodPressure)
CXC_FACTORY_ENTRY(setCholesterolReadings)
CXC_FACTORY_ENTRY(setEmploymentInfo)
CXC_FACTORY_ENTRY(setFamilyHistory)
CXC_FACTORY_ENTRY(setHealthConditions)
CXC_FACTORY_ENTRY(setImagingInfo)
CXC_FACTORY_ENTRY(setImmunizations)
CXC_FACTORY_ENTRY(setInsurance)
CXC_FACTORY_ENTRY(setMedications)
CXC_FACTORY_ENTRY(setReminder)
CXC_FACTORY_ENTRY(setSLMDLocations)
CXC_FACTORY_ENTRY(setSurgeryInfo)
CXC_FACTORY_ENTRY(setTherapyInfo)
CXC_FACTORY_ENTRY(setUnregisteredUser)
CXC_FACTORY_ENTRY(setUserBiographics)
CXC_FACTORY_ENTRY(setUserPhysicians)
CXC_FACTORY_ENTRY(setUserPreference)

//update commands
CXC_FACTORY_ENTRY(uptAddressInfo)
CXC_FACTORY_ENTRY(uptAdmit)
CXC_FACTORY_ENTRY(uptBiographics)
CXC_FACTORY_ENTRY(uptCareDirectives)
CXC_FACTORY_ENTRY(uptCompany)
CXC_FACTORY_ENTRY(uptConvertPc)
CXC_FACTORY_ENTRY(uptDiagnosis)
CXC_FACTORY_ENTRY(uptDischarge)
CXC_FACTORY_ENTRY(uptEmployment)
CXC_FACTORY_ENTRY(uptEncounter)
CXC_FACTORY_ENTRY(uptEncounterHcp)
CXC_FACTORY_ENTRY(uptEncounterLog)
CXC_FACTORY_ENTRY(uptExternalCode)
CXC_FACTORY_ENTRY(uptFacility)
CXC_FACTORY_ENTRY(uptGuarantor)
CXC_FACTORY_ENTRY(uptIdMap)
CXC_FACTORY_ENTRY(uptInsurance)
CXC_FACTORY_ENTRY(uptInsurancePlan)
CXC_FACTORY_ENTRY(uptLoa)
CXC_FACTORY_ENTRY(uptMiscIds)
CXC_FACTORY_ENTRY(uptName)
CXC_FACTORY_ENTRY(uptNok)
```

```

        CXC_FACTORY_ENTRY(uptPatient)
        CXC_FACTORY_ENTRY(uptPerson)
        CXC_FACTORY_ENTRY(uptPhone)
        CXC_FACTORY_ENTRY(uptPhysical)
        CXC_FACTORY_ENTRY(uptPreAdmit)
        CXC_FACTORY_ENTRY(uptTransfer)
    CXC_FACTORY_MAP_END()

CxcLCBrokerFactory::CxcLCBrokerFactory()
    :CXmlCommandFactory(CXC_FACTORY_BASE, CXC_FACTORY_COUNT)
{
}

////////////////////////////////////
// CxcLCBroker
////////////////////////////////////

CxcLCBroker::CxcLCBroker()
{
    m_pcoClient = NULL;
    m_emLast.clear();
}

void CxcLCBroker::setOwner(CComObjectRootBase * pcoOwner)
{
    m_pcoClient = (CoLCBroker *) pcoOwner;
    return;
}

//generates an audit Id, sticks to the audit table and return the Id.
inline
long CxcLCBroker::getAuditId()
{
    return m_pcoClient->getAuditIdGenerator()->generateId(0, 0, m_pcoClient->getHostName()
        , "LCBroker");
}

//generates an cpi Id
inline
long CxcLCBroker::getNewCpiId()
{
    return m_pcoClient->getCpiIdGenerator()->generateId();
}

//generates an enc Id
inline
long CxcLCBroker::getNewEncId()
{
    return m_pcoClient->getEncIdGenerator()->generateId();
}

//generates an unregistered_user Id
inline
long CxcLCBroker::getNewUnregUserId()
{
    return m_pcoClient->getUnregUserIdGenerator()->generateId();
}

//add a xml tag to the results document.
inline
void CxcLCBroker::openXmlTag(string strTagName, string strAttribute)
{
    CXMLElement xmlElTag, xmlElCurrent;

```

```
    if (m_pdocResults)
    {
        m_pdocResults->getCurrent(&xmlElCurrent);
        m_pdocResults->createElement(strTagName.c_str(), (LPCSTR) NULL, &xmlElTag);
        xmlElTag.setAttribute(XML_ATTR_TYPE, strAttribute.c_str());
        xmlElCurrent.addChild(&xmlElTag);
        m_pdocResults->pushCurrent(&xmlElTag);
    }
    return;
}

//close the xml tag
inline
void CxcLCBroker::closeXmlTag()
{
    if (m_pdocResults) m_pdocResults->popCurrent();
    return;
}

//add XML tag and its value to result document.
inline
void CxcLCBroker::addXmlChild(string strTagName, __variant_t vValue)
{
    CXMLElement xmlElCol, xmlElCurrent;
    string strValue ;

    strValue = (char *) _bstr_t (vValue);

    if (m_pdocResults)
    {
        m_pdocResults->getCurrent(&xmlElCurrent);
        m_pdocResults->createElement(strTagName.c_str(), strValue.c_str(), &xmlElCol);
        xmlElCol.setAttribute(XML_ATTR_TYPE, XML_TYPE_COLUMN);
        xmlElCurrent.addChild(&xmlElCol);
    }
    return;
}

//return the value of a field from the recordset as string
inline
string CxcLCBroker::getField(CSdoRecordset &rsRS, string strField)
{
    string strValue;
    rsRS.getField(strField.c_str(), strValue);
    return strValue.c_str();
}
```

```

////////////////////////////////////
// Include files
////////////////////////////////////

#include "stdafx.h"
#include "SldoBase.h"
#include "idGen.h"
#include "xcLCBroker.h"

////////////////////////////////////
// CxcLCBrokerModify
////////////////////////////////////

CxcLCBrokerModify::CxcLCBrokerModify()
{
}

////////////////////////////////////
//Checks for the existance of parameter "data"
bool CxcLCBrokerModify::parseCommand(CXmlDocument * pdoc)
{
    bool fSuccess;
    if (fSuccess = CXmlCommand::parseCommand(pdoc))
        m_docXmlCmd.attach(*pdoc);

    string strDummy;
    if (!(fSuccess = getParm("data", strDummy)))
        m_emLast.setError("Missing data parameter");

    return fSuccess;
}

////////////////////////////////////
// Pushes the data parameter element in stack and calls execute.
//
// [Call this method if the parameter name is data]
// [for custom parameter names, process parameter names and then call [execute]]
////////////////////////////////////

bool CxcLCBrokerModify::executeUpdate(CSdoRecordset &rsCmdRecSet, bool fGenerateAuditId)
{
    CXmlElement elRoot;
    m_docXmlCmd.getCurrent(&elRoot);          //get the root element

    CXmlElement elParm;
    elRoot.getFirst(&elParm);                 //get the param "data" element.

    m_docXmlCmd.pushCurrent(&elParm);         //Push the parm element to the stack.

    return execute(rsCmdRecSet, fGenerateAuditId);
}

/*****
FUNCTION : Generalised method to process xml data.

DESCRIPTION:
- fetches the data, rows, row tags
- calls virtual method parseParameter() for each ROW provided.
- sets all the parameters of the command.
- gets the audit id if required.
- gets database connection and executes the command.
*****/

```

USE:

- Call this method if u want to pass the data to stored procedure.
- Create the recordset object and set active command.
- Implement parameter parsing in parseParameter() method.

```

*****/
bool CxcLCBrokerModify::execute(CSdoRecordset &rsCmdRecSet, bool fGenerateAuditId)
{
    bool fSuccess = false;

    string strTag;
    CSdoConnection * pconn = NULL;

    m_emLast.clear();

    try
    {
        //Active command is expected to be set prior calling.

        //get the Sdo command pointer
        CSdoCommand * pcmdSdo = rsCmdRecSet.getActiveCommand();

        //get the connection
        pconn = m_pcoClient->getConnection();

        //start transaction
        pconn->beginTrans();

        //get the data parameter element from stack.
        CXmlElement elParm;
        m_docXmlCmd.getCurrent(&elParm);

        CXmlElement elRows;
        elParm.getFirst(&elRows); //get the "rows" Element
        elRows.getTag(strTag);
        if (strcmp(strTag.c_str(), "rows") != 0)
        {
            m_emLast.clear();
            m_emLast << "Expecting XML element \"rows\". Found \"" << strTag << "\".";
            return false;
        }

        CXmlElement elRow;
        bool fRows = elRows.getFirst(&elRow); //get the "row" Element

        while (fRows) //process for all "row" Elements
        {
            //get "row"tag
            elRow.getTag(strTag);
            if (strcmp(strTag.c_str(), "row") != 0)
            {
                m_emLast.clear();
                m_emLast << "Expecting XML element \"row\". Found \"" << strTag << "\".";
                fSuccess = false;
                break;
            }

            //Set all the parameters
            m_docXmlCmd.pushCurrent(&elRow);

            //Call the virtual method to do any parameter validation.
            if (!parseParameters())
            {
                fSuccess = false;
                break;
            }
        }
    }
}

```

```

        pcmdSdo->clearParms();
        if (pcmdSdo->setParms(m_docXmlCmd) == false)
        {
            string strError;
            pcmdSdo->getLastError(strError);
            m_emLast.setError(strError.c_str());
            fSuccess = false;
            break;
        }
        m_docXmlCmd.popCurrent();

        //Generate and set AuditID if applicable.
        if (fGenerateAuditId)
        {
            // generate audit id
            long lNewId = getAuditId();
            pcmdSdo->setParm("audit_id", _variant_t(lNewId));
        }

        // update the table
        if (!(fSuccess = pconn->execute(rsCmdRecSet)))
        {
            m_emLast.setError(pconn->getLastError());
            fSuccess = false;
            break;
        }

        //fetch next row to update.
        fRows = elRows.getNext(&elRow);
    }
}
catch(_com_error & e)
{
    m_emLast.setError(e);
    fSuccess = false;
}
catch(...)
{
    m_emLast.setError("Unkown exception raised.[CxcLCBrokerModify::execute
(parameters)]");
    fSuccess = false;
}

//commit or Roll back the transaction.
if (pconn)
{
    if (fSuccess)    pconn->commitTrans();
    else            pconn->rollbackTrans();
}

return fSuccess;
}

/*****
FUNCTION : Generalised method to process xml data.

DESCRIPTION:
- fetches the data, rows, row tags
- calls virtual method parseParameter() for eah ROW provided
- calls virtual method processData() for each ROW provided.

USE:
- Call this method to loop throught all the data ROWS provided.

```

- Implement the data processing logic in processData() method.
- Implement parameter parsing in parseParameter() method.

```

*****/
bool CxcLCBrokerModify::execute()
{
    bool fSuccess = false;

    string strTag;
    m_emLast.clear();

    try
    {
        CXmlElement elRoot;
        m_docXmlCmd.getCurrent(&elRoot);           //get the root element

        CXmlElement elParm;
        elRoot.getFirst(&elParm);                 //get the param "data" element.

        CXmlElement elRows;
        elParm.getFirst(&elRows);                 //get the "rows" Element
        elRows.getTag(strTag);
        if (strcmp(strTag.c_str(), "rows") != 0)
        {
            m_emLast.clear();
            m_emLast << "Expecting XML element \"rows\". Found \"" << strTag << "\".";
            return false;
        }

        CXmlElement elRow;
        bool fRows = elRows.getFirst(&elRow);     //get the "row" Element

        while (fRows)                             //process for all "row" Elements
        {
            //get "row" tag
            elRow.getTag(strTag);
            if (strcmp(strTag.c_str(), "row") != 0)
            {
                m_emLast.clear();
                m_emLast << "Expecting XML element \"row\". Found \"" << strTag << "\".";
                fSuccess = false;
                break;
            }

            //push this row element to stack.
            m_docXmlCmd.pushCurrent(&elRow);

            //Call the virtual method to do any parameter validation.
            if (!parseParameters())
            {
                fSuccess = false;
                break;
            }

            //Call the virtual method to do the data processing.
            if (!processData())
            {
                fSuccess = false;
                break;
            }

            //fetch next row to update.
            fRows = elRows.getNext(&elRow);
        }
    }
    catch(_com_error & e)

```



```
{
    m_emLast.setError(e);
    fSuccess = false;
}
catch(...)
{
    m_emLast.setError("Unkown exception raised.[CxcLCBrokerModify::execute]");
    fSuccess = false;
}

return fSuccess;
}

/*****
FUNCTION : Return the parameter value by parameter name.
DESCRIPTION:
    - fetches the data value by the provided tag
USE:
*****/
string CxcLCBrokerModify::getParameterValue(string strName)
{
    string strValue;
    CXMLElement elRow, elParm;

    //Get the Current Row tag from the stack.
    m_docXmlCmd.getCurrent(&elRow);

    strValue = "";
    if (elRow.getFirstItem(strName.c_str(), &elParm))
        elParm.getText(strValue);

    return strValue;
}
```

```

#include "stdafx.h"
#include "XmlParser.h"
#include "xmlCommand.h"

/*****
Class: CXmlCommandFactory

Purpose: Serves as base class for specialized command factories. Parses
xml commands and instantiates specialized CXmlCommand based on
the command name. Derived classes overrid createCommandByName()
to create specific commands.

*****/

/*****
Class: CXmlCommand

Purpose: Serves as base class for specialized commands. Derived classes
can override parseCommand() to handle anomalies in xml command
format and execCommand() to carry out the purpose of the command.

*****/

/*****
Class: CXmlCmdBase

Purpose: Provides some rudimentary functionality common to all derivatives
of CXmlCommand and CXmlCommandFactory

*****/

////////////////////////////////////

CXmlCommand::CXmlCommand()
{
    m_fConnectRequired = true;
}

CXmlCommand::~CXmlCommand()
{
    if (m_fOwnsResults && m_pdocResults != NULL)
        delete m_pdocResults;
}

/*****
FUNCTION: initThis

CLASS: CXmlCommand

DESCRIPTION: Object initializer. Empties command name and parm list.

*****/
void CXmlCommand::initThis()
{
    m_strCommand = "";
    m_pdocResults = NULL;
    m_pcoOwner = NULL;
    m_fOwnsResults = false;
    m_mapParms.clear();
}

CXmlDocument * CXmlCommand::getResults(bool fTakeOwnership)
{
    m_fOwnsResults = !fTakeOwnership;
    return m_pdocResults;
}

```

```

void CXmlCommand::setOwner(CComObjectRootBase * pcoOwner)
{
    m_pcoOwner = pcoOwner;
}

/*****
    FUNCTION: parseCommand

    CLASS: CXmlCommand

    DESCRIPTION: This function can be overridden by derived classes. Base implementation
                 moves xml command name and parm name/value pairs into member variables.

    PARAMETERS: pdoc - A CXmlDocument object containing parsed xml.

    RETURNS: true on success.
*****/
bool CXmlCommand::parseCommand(CXmlDocument * pdoc)
{
    bool fSuccess = false;

    string strTag;
    string strAttr;
    string strValue;

    initThis();

    try
    {
        CXmlElement xmlRoot;
        pdoc->getCurrent(&xmlRoot);

        // root tag must be "command"
        xmlRoot.getTag(strTag);
        if (strcmp(strTag.c_str(), "command") != 0)
        {
            m_emLast.setError("Document tag must be \"command\".");
            throw E_FAIL;
        }

        // the command must have a "name" attribute
        if (xmlRoot.getAttribute("name", strAttr) == false)
        {
            m_emLast.setError("Command tag must have a \"name\" attribute.");
            throw E_FAIL;
        }

        m_strCommand = strAttr;

        // get each parameter
        CXmlElement xmlParm;
        bool fFound = xmlRoot.getFirst(&xmlParm);
        while (fFound)
        {
            // parms have a tag of "parm"
            xmlParm.getTag(strTag);
            if (strcmp(strTag.c_str(), "parm") == 0)
            {
                // parms must have an attribute of "name"
                if (xmlParm.getAttribute("name", strAttr) == false)
                {
                    m_emLast.setError("Parm tags must have a \"name\" attribute.");
                    throw E_FAIL;
                }

                TOUPPER(strAttr);
            }
        }
    }
}

```

```

        xmlParm.getText(strValue);
        m_mapParms[strAttr] = strValue;
    }

    // next parm
    fFound = xmlRoot.getNext(&xmlParm);
}

    fSuccess = true;
}
catch(...)
{
}

return fSuccess;
}

/*****
FUNCTION: execCommand

CLASS: CXmlCommand

DESCRIPTION: This function should be overridden by derived classes. It can
            be used to carry out the function of a command.

PARAMETERS:

RETURNS: true on success

*****/
bool CXmlCommand::execCommand()
{
    return true;
}

/*****
FUNCTION: getParm

CLASS: CXmlCommand

DESCRIPTION: retrieves a parm's value by name.

PARAMETERS: pszParmName - name of parm to retrieve
            strParmValue - string receiving the parm's value.

RETURNS:

*****/
bool CXmlCommand::getParm(LPCSTR pszParmName, string & strParmValue)
{
    bool fFound = false;
    map<string, string>::iterator itParms;

    string strKey = pszParmName;
    TOUPPER(strKey);

    itParms = m_mapParms.find(strKey);
    if (fFound = itParms != m_mapParms.end())
        strParmValue = (*itParms).second;
    else
        strParmValue = "";

    return fFound;
}

//

```

```

CXmlCommandFactory::CXmlCommandFactory(FACTORYMAP * pmapFactories, int nNumFactoryEntries)
{
    m_pmapFactories = pmapFactories;
    m_nNumberFactoryEntries = nNumFactoryEntries;
}

```

```

/*****

```

```

    FUNCTION: createCommand

```

```

        CLASS: CXmlCommandFactory

```

```

    DESCRIPTION: Creates a command based on the passed in xml data.

```

```

    PARAMETERS: bstrXmlCommand - contains xml data that is a command

```

```

    RETURNS: the new CxlCommand

```

```

*****/

```

```

CXmlCommand * CXmlCommandFactory::createCommand(BSTR bstrXmlCommand)

```

```

{
    string strCmd = (char *) _bstr_t(bstrXmlCommand);
    return createCommand(strCmd.c_str());
}

```

```

/*****

```

```

    FUNCTION: createCommand

```

```

        CLASS: CXmlCommandFactory

```

```

    DESCRIPTION: Creates a command based on the passed in xml data.

```

```

    PARAMETERS: pszXmlCommand - contains xml data that is a command

```

```

    RETURNS: the new CxlCommand

```

```

*****/

```

```

CXmlCommand * CXmlCommandFactory::createCommand(LPCSTR pszXmlCommand)

```

```

{
    HRESULT hr = S_OK;

    bool fSuccess = false;

    CXmlCommand * pcmdXml = NULL;
    CXmlDocument * pdocXml = NULL;

    try
    {
        pdocXml = new CXmlDocument(pszXmlCommand);
        if (!pdocXml->isReady())
        {
            string strError;
            pdocXml->getParserError(strError);
            m_emLast.clear();
            m_emLast << "XML Parser Error [" << strError << "];"
        }
        else
            pcmdXml = createCommand(pdocXml);
    }
    catch(HRESULT hrError)
    {
        m_emLast.clear();
        m_emLast << "Unable to create IStream. hr = [" << std::hex << hrError << "];"
    }

    return pcmdXml;
}

```

```

/*****
    FUNCTION: createCommand

    CLASS: CXmlCommandFactory

    DESCRIPTION: Creates a command based on the passed in xml data.

    PARAMETERS: pdoc - contains xml data that is a command

    RETURNS: the new CxlCommand
*****/
CXmlCommand * CXmlCommandFactory::createCommand(CXmlDocument * pdocXML)
{
    bool fSuccess = false;

    CXmlCommand * pcmdXml = NULL;

    string strTag;
    string strAttr;

    try
    {
        // get the command
        CXmlElement xmlRoot;
        pdocXML->getCurrent(&xmlRoot);
        xmlRoot.getTag(strTag);
        if (strcmp(strTag.c_str(), "command") != 0)
        {
            m_emLast.setError("Document level tag is not \"command\"");
            fSuccess = false;
        }
        else
            fSuccess = true;

        if (fSuccess)
        {
            try
            {
                xmlRoot.getAttribute("name", strAttr);
            }
            catch(...)
            {
                m_emLast.setError("XML is missing command name attribute.");
                fSuccess = false;
            }

            if (fSuccess)
            {
                // create the command, last error is set if unsuccessful
                pcmdXml = createCommandByName(strAttr.c_str());
                if (fSuccess = (pcmdXml != NULL))
                    fSuccess = pcmdXml->parseCommand(pdocXML);
            }
        }
    }
    catch(_com_error & e)
    {
        m_emLast.setError(e);
        fSuccess = false;
    }
    catch(...)
    {
        m_emLast.setError("Unknown exception raised.");
        fSuccess = false;
    }
}

```

```

    if (!fSuccess)
    {
        if (pcmdXml != NULL)
        {
            string strError;
            pcmdXml->getLastError(strError);
            m_emLast.setError(strError.c_str());
            delete pcmdXml;
            pcmdXml = NULL;
        }
    }

    return pcmdXml;
}

/*****
FUNCTION: createCommandByName

CLASS: CXmlCommandFactory

DESCRIPTION: Searches factory map provided by derived class and calls a
             found factory. Derived class can override this function.

PARAMETERS: pszCommandName - the name of the command to instantiate.

RETURNS: the newly created CXmlCommand.
*****/
CXmlCommand * CXmlCommandFactory::createCommandByName(LPCSTR pszCommandName)
{
    CXmlCommand * pcmdXml = NULL;

    FACTORYMAP * pfactoryCmd = (FACTORYMAP *) bsearch((void *) pszCommandName,
        (void *) m_pmapFactories,
        m_nNumberFactoryEntries,
        sizeof(FACTORYMAP),
        compareEntry);

    if (pfactoryCmd != NULL)
        pcmdXml = (*(pfactoryCmd->m_pfuncFactory))();

    if (pcmdXml == NULL)
    {
        m_emLast.clear();
        m_emLast << "[" << pszCommandName << "]" is not a recognizable command.";
    }

    return pcmdXml;
}

int CXmlCommandFactory::compareEntry(const void * pszKey, const void * pFactoryEntry)
{
    return strcmp((const char *) pszKey, ((FACTORYMAP *)pFactoryEntry)->m_pszCommand);
}

```

```
#ifndef _XmlParser_h
#define _XmlParser_h
```

```

/*****
Dependencies - the following should be stdafx.h

```

```
#import "msxml.dll"
```

```
#include <vector>
```

```
#include <string>
```

```
using namespace std;
```

```
*****/
```

```

/*****

```

This file implements a wrapper around msxml.dll. Msxml.dll is an in proc COM xml parser. The wrapper simplifies the view of an xml tree with the following.

1. Encapsulates document construction with constructors that take IStream, LPCSTR, or BSTR. IPersistStream and IStream manipulation occurs internally.
2. Provides an element stack on the document, allowing a caller to push a new root element. This allows the document to be passed to recursive routines working down the element tree.
3. Collapses an element tag and its child text element into one element with a tag and a text attribute.
4. Provides child iteration functions off of the element. This hides the extra steps needed to manipulate the COM object enumerators.

```
*****/
```

```

////////////////////////////////////
// CXmlElement

```

```
class CXmlElement
```

```
{
    friend class CXmlDocument;
```

```
protected:
```

```
    MSXML::IXMLDOMElementPtr    m_pIElement;
    int                          m_nItemIdx;
```

```
public:
```

```

    CXmlElement();
    CXmlElement(const CXmlElement & elRight);
    CXmlElement(MSXML::IXMLDOMElementPtr & pIElement);
    ~CXmlElement();
    CXmlElement & operator=(MSXML::IXMLDOMElementPtr pIElement);
    CXmlElement & operator=(const CXmlElement & elRight);
    void getTag(string & strTag);
    bool getAttribute(LPCSTR pszAttribute, string & strAttribute);
    void setAttribute(LPCSTR pszAttribute, _variant_t vValue);
    void setAttribute(LPCSTR pszAttribute, string & strValue);
    void setAttribute(LPCSTR pszAttribute, long lValue);
    bool setText(LPCSTR strText, CXmlDocument * pxmldoc = NULL);
    void getText(string & strText);
    bool addChild(CXmlElement * pxmldoc);
    bool getFirst(CXmlElement * pxmldoc);
    bool getNext(CXmlElement * pxmldoc);
    bool getFirstItem(LPCSTR pszTag, CXmlElement * pxmldoc);
    bool getNextItem(LPCSTR pszTag, CXmlElement * pxmldoc);
    long toLong();
};

```

```

////////////////////////////////////
// CXmlDocument

```



```

#define NOVALUE ((LPCSTR)NULL)
typedef vector<CXmlElement> CStackElements;
class CXmlDocument
{
    friend class CXmlElement;

protected:
    MSXML::IXMLDOMDocumentPtr    m_pIDoc;
    CStackElements               m_stackCurrent;
    bool                         m_fReady;
    bool                         m_fUpperCaseTags;

public:
    CXmlDocument();
    CXmlDocument(LPCSTR pszXml);
    CXmlDocument(IStream * pIStream);
    ~CXmlDocument();
    bool attach(MSXML::IXMLDOMDocumentPtr spXmlDoc);
    bool attach(CXmlDocument & docXml);
    bool isReady();
    bool loadDocument(LPCSTR pszXml);
    bool getParserError(string & strError);
    void createElement(LPCSTR pszTag, LPCSTR pszValue, CXmlElement * pxmlElement);
    void createElement(LPCSTR pszTag, long lValue, CXmlElement * pxmlElement);
    void createElement(LPCSTR pszTag, _variant_t vValue, CXmlElement * pxmlElement);
    bool addChild(CXmlElement * pxmlElement);
    bool addChild(LPCSTR pszTag, LPCSTR pszValue = NULL, CXmlElement * pxmlElement = NULL) ✓
    ;
    bool addChild(LPCSTR pszTag, _variant_t vValue, CXmlElement * pxmlElement = NULL);
    bool getXML(char * pszXml, long * plBuffSize);
    bool getXML(string & strXml);
    bool getRoot(CXmlElement * pxmlElement);
    void pushCurrent(CXmlElement * pxmlElement);
    void popCurrent();
    void getCurrent(CXmlElement * pxmlElement);
    void setTagCaseToLower();
};

inline void CXmlElement::setAttribute(LPCSTR pszAttribute, _variant_t vValue)
{
    _ASSERT(m_pIElement != NULL);
    m_pIElement->setAttribute(_bstr_t(pszAttribute), vValue);
}

inline void CXmlElement::setAttribute(LPCSTR pszAttribute, string & strValue)
{
    setAttribute(pszAttribute, _variant_t(strValue.c_str()));
}

inline void CXmlElement::setAttribute(LPCSTR pszAttribute, long lValue)
{
    setAttribute(pszAttribute, (char *) (_bstr_t) _variant_t((long)lValue));
}

inline void CXmlElement::getTag(string & strTag)
{
    _ASSERT(m_pIElement != NULL);
    strTag = (char *) m_pIElement->tagName;
    return;
}

inline CXmlElement & CXmlElement::operator=(MSXML::IXMLDOMElementPtr pIElement)
{
    m_pIElement = pIElement;
    m_nItemId = 0;
    return *this;
}

```

```
}

inline CXmlElement & CXmlElement::operator=(const CXmlElement & elRight)
{
    m_pIElement = elRight.m_pIElement;
    m_nItemIdx = 0;
    return *this;
}

inline bool CXmlElement::addChild(CXmlElement * pxmlElement)
{
    _ASSERT(m_pIElement != NULL);
    return m_pIElement->appendChild(pxmlElement->m_pIElement) != NULL;
}

inline bool CXmlElement::getFirst(CXmlElement * pxmlElement)
{
    _ASSERT(m_pIElement != NULL);
    m_nItemIdx = 0;
    return getNext(pxmlElement);
}

inline bool CXmlElement::getFirstItem(LPCSTR pszTag, CXmlElement * pxmlElement)
{
    _ASSERT(m_pIElement != NULL);
    m_nItemIdx = 0;
    return getNextItem(pszTag, pxmlElement);
}

inline long CXmlElement::toLong()
{
    _ASSERT(m_pIElement != NULL);

    string strText;
    getText(strText);
    if (strText.size())
        return atol(strText.c_str());
    else
        return 0;
}

inline void CXmlDocument::setTagCaseToLower()
{
    m_fUpperCaseTags = false;
}

inline bool CXmlDocument::isReady()
{
    return m_fReady;
}

inline bool CXmlDocument::addChild(CXmlElement * pxmlElement)
{
    _ASSERT(m_pIDoc != NULL);
    return m_stackCurrent[0].addChild(pxmlElement);
}

inline bool CXmlDocument::getRoot(CXmlElement * pxmlElement)
{
    bool fSuccess;

    if (fSuccess = m_pIDoc != NULL && m_fReady && m_stackCurrent.size() > 0)
        *pxmlElement = m_stackCurrent[0];

    return fSuccess;
}
```

```
inline bool CXmlDocument::attach(CXmlDocument & docXml)
{
    CXmlElement elCurrent;
    docXml.getCurrent(&elCurrent);
    bool fSuccess = attach(docXml.m_pIDoc);
    if (fSuccess)
        pushCurrent(&elCurrent);
    return fSuccess;
}

#endif
```

```
#include "stdafx.h"
#include "zipUtil.h"
```

```
CZipUtil::CZipUtil()
{
}
```

```
CZipUtil::~CZipUtil()
{
}
```

```
CZipUtilXceed::CZipUtilXceed()
{
    HRESULT hr = m_spZip.CreateInstance(__uuidof(XZip::XceedZip));
    if (FAILED(hr))
    {
        CLogMsgEvent msg(LCEV_GENERIC, SVRTY_WARNING);
        msg << "Unable to Instantiate XceedZip. Error = [0x" << std::hex << hr << "]";
        msg.Post(_logAll);
    }
}
```

```
bool CZipUtilXceed::unzipFile(LPCSTR pszZipFile, LPCSTR pszTargetDir, unsigned short nFlags)
{
    bool fSuccess = true;

    m_spZip->ZipFilename = pszZipFile;
    m_spZip->UnzipToFolder = pszTargetDir;
    m_spZip->PreservePaths = (nFlags & ZF_UseDirectoryNames) ? TRUE : FALSE;
    m_spZip->SkipIfExists = (nFlags & ZF_OverWrite) ? FALSE : TRUE;
    XZip::xcdError eErrorCode = m_spZip->Unzip();
    if (eErrorCode != XZip::xerSuccess)
    {
        CLogMsgEvent msg(LCEV_GENERIC, SVRTY_WARNING);
        msg << "Error occurred while unzipping file [" << pszZipFile << "] to directory [" << pszTargetDir << "]. Xceed Error Code = [" << (long) eErrorCode << "];";
        msg.Post(_logAll);
        fSuccess = false;
    }

    return fSuccess;
}
```

```
#include "stdafx.h"
#include "dialer.h"

////////////////////////////////////
// CDialer
CDialer::CDialer()
{
    m_hconn = NULL;
    m_hwndOwner = NULL;
}

CDialer::CDialer(LPCSTR pszPhoneEntry, LPCSTR pszPhoneBook)
{
    m_hwndOwner = NULL;
    m_hconn = NULL;
    if (pszPhoneEntry != NULL)
        m_strPhoneEntry = pszPhoneEntry;
    if (pszPhoneBook)
        m_strPhoneBook = pszPhoneBook;
}

CDialer::~CDialer()
{
}

bool CDialer::isConnected()
{
    return m_hconn != NULL;
}

////////////////////////////////////
// CDialerRAS

static char * m_pszConnStates[] = {
    "OpenPort",
    "PortOpened",
    "ConnectDevice",
    "DeviceConnected",
    "AllDevicesConnected",
    "Authenticate",
    "AuthNotify",
    "AuthRetry",
    "AuthCallback",
    "AuthChangePassword",
    "AuthProject",
    "AuthLinkSpeed",
    "AuthAck",
    "ReAuthenticate",
    "Authenticated",
    "PrepareForCallback",
    "WaitForModemReset",
    "WaitForCallback",
    "Projected",
    "StartAuthentication",
    "CallbackComplete",
    "LogonNetwork",
    "SubEntryConnected",
    "SubEntryDisconnected",
    "Interactive = RASCS_PAUSED",
    "RetryAuthentication",
    "CallbackSetByCaller",
    "PasswordExpired",
    "InvokeEapUI",
    "Connected",
    "Disconnectedt"};

#define CONN_STATES_MAX (sizeof(m_pszConnStates) / sizeof(char *))
```

```
CDialerRAS::CDialerRAS()
{
}

CDialerRAS::CDialerRAS(LPCSTR pszPhoneEntry, LPCSTR pszPhoneBook)
:CDialer(pszPhoneEntry, pszPhoneBook)
{
}

unsigned long CDialerRAS::getNotificationMessId()
{
    unsigned long nMessId = RegisterWindowMessageA(RASDIALEVENT);
    if (nMessId == 0)
        nMessId = WM_RASDIALEVENT;
    return nMessId;
}

void CDialerRAS::logNotification(unsigned long lRasStatus, unsigned long dwError)
{
    if (dwError == 0)
    {
        CLogMsgEvent msg(LCEV_GENERIC, SVRTY_INFO);
        if (lRasStatus < CONN_STATES_MAX)
        {
            msg << "RAS Dialing State = [" << m_pszConnStates[lRasStatus] << "];
            msg.Post(_logAll);
        }

        if (lRasStatus == RASCS_Authenticated)
            PostMessage(m_hwndOwner, WMUSER_CONNECTED, 0, 0);
    }
    else
    {
        CLogMsgEvent msg(LCEV_GENERIC, SVRTY_WARNING);
        char pszError [256];
        RasGetErrorString(dwError, pszError, 256);
        msg << "RAS Error = [" << pszError << "];
        msg.Post(_logAll);

        PostMessage(m_hwndOwner, WMUSER_CONNECTED, 0, dwError);
    }
}

bool CDialerRAS::getEntryPhoneNumber(LPCSTR pszEntry, string & strPhoneNumber)
{
    stringstream      strmPhoneNumber;
    RASENTRY * pentryRas;
    DWORD dwBuffSize = sizeof(RASENTRY);

    unsigned char * pBuff = new unsigned char [dwBuffSize];
    memset(pBuff, 0, dwBuffSize);
    pentryRas = (RASENTRY *) pBuff;
    pentryRas->dwSize = sizeof(RASENTRY);

    DWORD dwErr = RasGetEntryProperties(NULL, pszEntry, pentryRas, &dwBuffSize, NULL, 0);

    // if buffer was to small try again
    if (dwErr)
    {
        delete [] pBuff;
        pBuff = new unsigned char [dwBuffSize];
        memset(pBuff, 0, dwBuffSize);
        pentryRas = (RASENTRY *) pBuff;
        pentryRas->dwSize = sizeof(RASENTRY);
        dwErr = RasGetEntryProperties(NULL, pszEntry, pentryRas, &dwBuffSize, NULL, 0);
    }
}
```

```
// got a real error
if (dwErr)
{
    char pszError [256];
    RasGetErrorString(dwErr, pszError, 256);
    CLogMsgEvent msg(LCEV_GENERIC, SVRTY_WARNING);
    msg << "getEntryPhoneNumber() Error = [" << pszError << "];
    msg.Post(_logAll);
}
else
{
    if (pentryRas->dwfOptions & RASEO_UseCountryAndAreaCodes)
        strmPhoneNumber << pentryRas->dwCountryCode << pentryRas->szAreaCode;
    strmPhoneNumber << pentryRas->szLocalPhoneNumber;
    strPhoneNumber = strmPhoneNumber.str();
}

if (pBuff != NULL)
    delete [] pBuff;

return dwErr == 0;
}

bool CDialerRAS::connect()
{
    // todo: this is weird way to dial a phone book entry. I could only get connects
    // when I used a phone number only. So this routine retrieves the user, password, and
    // phone number from the default phone book and then does a modem dial to get
    // connected.
    // A better way to do it would be to just use the entry name in the RasDial() call but
    // I couldn't get it to authenticate.

    RASDIALPARAMS  parmsRas;
    string strPhoneNumber;
    BOOL fPassword;

    // get user and password for the entry
    memset(&parmsRas, 0, sizeof(parmsRas));
    parmsRas.dwSize = sizeof(parmsRas);
    strcpy(parmsRas.szEntryName, m_strPhoneEntry.c_str());
    DWORD dwError = RasGetEntryDialParams(NULL, &parmsRas, &fPassword);
    if (dwError)
        goto errorConnect;

    if (!getEntryPhoneNumber(parmsRas.szEntryName, strPhoneNumber))
        return false;
    strcpy(parmsRas.szPhoneNumber, strPhoneNumber.c_str());
    parmsRas.szEntryName[0] = 0;
    parmsRas.szDomain[0] = 0;
    if (dwError = RasDial(NULL, NULL, &parmsRas, -1, m_hwndOwner, &m_hconn))
        goto errorConnect;

    return true;

errorConnect:
    char pszError [256];
    RasGetErrorString(dwError, pszError, 256);
    CLogMsgEvent msg(LCEV_GENERIC, SVRTY_WARNING);
    msg << "RasDial Error = [" << pszError << "];
    msg.Post(_logAll);
    return false;
}

bool CDialerRAS::disconnect()
{
    DWORD dwError = NULL;
}
```

```
bool fSuccess = true;
if (m_hconn != NULL)
{
    if (dwError = RasHangUp(m_hconn))
    {
        char pszError [256];
        RasGetErrorString(dwError, pszError, 256);
        CLogMsgEvent msg(LCEV_GENERIC, SVRTY_WARNING);
        msg << "RasHangUp Error = [" << pszError << "]";
        msg.Post(_logAll);
        fSuccess = false;
    }
}

m_hconn = NULL;

return fSuccess;
}

////////////////////////////////////
// CDialerWinInet
CDialerWinInet::CDialerWinInet()
{
}

CDialerWinInet::CDialerWinInet(LPCSTR pszPhoneEntry, LPCSTR pszPhoneBook)
:CDialer(pszPhoneEntry, pszPhoneBook)
{
}

bool CDialerWinInet::connect()
{
    bool fSuccess;

    DWORD dwError = InternetDial(m_hwndOwner, "TXU", INTERNET_AUTODIAL_FORCE_UNATTENDED,
        (unsigned long *) &m_hconn, 0L);
    if (!(fSuccess = dwError == 0))
    {
        CLogMsgEvent msg (LCEV_GENERIC, SVRTY_WARNING);
        msg << "InternetDial Error = [" << std::hex << dwError << std::dec << "]";
        msg.Post(_logAll);
    }

    PostMessage(m_hwndOwner, WMUSER_CONNECTED, 0, dwError);

    return fSuccess;
}

bool CDialerWinInet::disconnect()
{
    bool fSuccess = true;
    DWORD dwError = 0;

    if (m_hconn != NULL)
    {
        dwError = InternetHangUp((unsigned long) m_hconn, NULL);
        if (!(fSuccess = dwError == 0))
        {
            CLogMsgEvent msg (LCEV_GENERIC, SVRTY_WARNING);
            msg << "InternetHangUp Error = [" << std::hex << dwError << std::dec << "]";
            msg.Post(_logAll);
        }
    }

    return fSuccess;
}
```



```
#ifndef _dialer_h
#define _dialer_h

#include "ras.h"
#include "wininet.h"

class CDialer
{
protected:
    HRASCONN    m_hconn;

public:
    HWND        m_hwndOwner;
    string      m_strPhoneBook;
    string      m_strPhoneEntry;

public:
    CDialer();
    CDialer(LPCSTR pszPhoneEntry, LPCSTR pszPhoneBook = NULL);
    virtual ~CDialer();
    virtual bool connect() = 0;
    virtual bool disconnect() = 0;
    virtual bool isConnected();
};

class CDialerRAS : public CDialer
{
public:
    CDialerRAS();
    CDialerRAS(LPCSTR pszPhoneEntry, LPCSTR pszPhoneBook = NULL);
    unsigned long getNotificationMessId();
    void logNotification(unsigned long lRasStatus, unsigned long dwError);
    virtual bool connect();
    virtual bool disconnect();
    bool getEntryPhoneNumber(LPCSTR pszEntry, string & strPhoneNumber);
};

class CDialerWinInet : public CDialer
{
public:
    CDialerWinInet();
    CDialerWinInet(LPCSTR pszPhoneEntry, LPCSTR pszPhoneBook = NULL);
    virtual bool connect();
    virtual bool disconnect();
};

#endif
```

```

// Encryptor.cpp: implementation of the CEncryptor class.
//
//
//
#include "StdAfx.h"
#include "Encryptor.h"
#include <time.h>

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//
// Construction/Destruction
//

CEncryptor::CEncryptor()
{
    m_strDefaultKey = "phepmagi";
}

/*****
FUNCTION: Encrypt

CLASS: CEncryptor

DESCRIPTION: Encrypts an ascii string into a series of ascii hex digits.

PARAMETERS: pszIn - pointer to string to encrypt
             pszKey - encryption key, this parameter may be null, in which
                    case a default encryption key is used.
             strOut - reference to a string that will receive the encryption
                    results.

RETURN3: true on success
        false on error
*****/
bool CEncryptor::Encrypt(LPCTSTR pszIn, LPCSTR pszKey, string & strOut)
{
    strOut = "";

    string strKey = pszKey == NULL ? m_strDefaultKey : pszKey;
    int nKeyLen = strKey.size();
    int nKeyPos = -1;

    srand((unsigned)time(NULL));
    int nOffset = rand() % 255;

    char pszBuff [4];
    sprintf(pszBuff, "%02x", nOffset);
    strOut += pszBuff;

    char chKey;

    for (int i = 0; pszIn[i] != 0; i++)
    {
        int nSrcAscii = (pszIn[i] + nOffset) % 255;

        if (nKeyPos < nKeyLen - 1)
        {
            nKeyPos++;
            chKey = strKey[nKeyPos];
        }
        else
        {
            nKeyPos = -1;
        }
    }
}

```

```

        chKey = 0;
    }

    nSrcAscii ^= chKey;
    sprintf(pszBuff, "%02x", nSrcAscii);
    strOut += pszBuff;
    nOffset = nSrcAscii;
}

return true;
}

/*****
FUNCTION: Decrypt

CLASS: CEncryptor

DESCRIPTION: Given encrypted data, decrypts back to its original form.

PARAMETERS: pszIn    - pointer to encrypted data.
             pszKey    - pointer to key that was used to encrypt the data. This
                        may be NULL in which case a default key is used.
             strOut    - reference to a string that will receive the decrypted
                        data.

RETURNS: true - no errors
        false - an error occurred
*****/
bool CEncryptor::Decrypt(LPCTSTR pszIn, LPCSTR pszKey, string & strOut)
{
    string strKey = pszKey == NULL ? "" : pszKey;
    if (strKey.size() == 0)
        strKey = m_strDefaultKey;

    strOut = "";

    int nSrcPos = 2;
    int nKeyPos = -1;
    string strSrc = pszIn;
    int nSrcLen = strSrc.size();
    int nKeyLen = strKey.size();
    int nSrcAscii = 0;
    int nTmpSrcAscii = 0;

    int nOffset;
    if (AsciiHexToInt(strSrc.substr(0, 2), &nOffset))
        return false;

    char chKey;

    do
    {
        if (AsciiHexToInt(strSrc.substr(nSrcPos, 2), &nSrcAscii))
            return false;

        if (nKeyPos < nKeyLen - 1)
        {
            nKeyPos += 1;
            chKey = strKey[nKeyPos];
        }
        else
        {
            nKeyPos = -1;
            chKey = 0;
        }
    }

```

```

        nTmpSrcAscii = nSrcAscii ^ chKey;

        if (nTmpSrcAscii <= nOffset)
            nTmpSrcAscii += 255 - nOffset;
        else
            nTmpSrcAscii -= nOffset;

        strOut += (char)nTmpSrcAscii;
        nOffset = nSrcAscii;
        nSrcPos += 2;
    } while (nSrcPos < nSrcLen);

    return true;
}

/* *****
FUNCTION: AsciiHexToInt

CLASS: CEncryptor

DESCRIPTION: Helper function that takes a string of ascii hex digits
            (ie. "EF34DC") and returns the binary decimal representation.

PARAMETERS: pszString - ascii hex digits to convert
            pnAnswer - pointer to an int that will receive the conversion.

RETURNS: true on error
        false on success
***** */
short CEncryptor::AsciiHexToInt(LPCTSTR pszString, int * pnAnswer)
{
    int nPlaces = strlen(pszString) - 1;

    short    wError = FALSE;
    char      cWork;
    int nAnswer = 0;

    for (int i = 0; !wError && (cWork = pszString[i]) != 0; i++)
    {
        cWork = toupper(cWork);
        if (!isdigit(cWork))
        {
            cWork -= 'A' - 10;
            if (cWork < 0 || cWork > 15)
                wError = TRUE;
        }
        else
            cWork &= 0x0f;

        if (nPlaces)
            nAnswer += cWork * (nPlaces-- * 16);
        else
            nAnswer += cWork;
    }

    *pnAnswer = nAnswer;
    return wError;
}

```

```
/*//////////////////////////////////////////
//////////////////////////////////////////
Encrypt/ Decrypt Rotines
//////////////////////////////////////////
```

Dependencies :

```
#include <string>
#include <list>
#include <fstream>
#include <sstream>
using namespace std;
```

```
//////////////////////////////////////////
//////////////////////////////////////////*/
```

```
#ifndef _ENCRYPTOR_H
#define _ENCRYPTOR_H
```

```
#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
```

```
class CEncryptor
```

```
{
```

```
protected:
```

```
    string m_strDefaultKey;
```

```
    short AsciiHexToInt( LPCTSTR pszString, int* pnAnswer );
```

```
    short AsciiHexToInt( strings strIn, int* pnAnswer )
```

```
        {return AsciiHexToInt(strIn.c_str(), pnAnswer);}
```

```
public:
```

```
    CEncryptor();
```

```
    bool Encrypt(LPCTSTR pszIn, LPCSTR psKey, string & strOut);
```

```
    bool Decrypt(LPCTSTR pszIn, LPCSTR psKey, string & strOut);
```

```
};
```

```
#endif // _ENCRYPTOR_H
```

```
#include "stdafx.h"
#include "filenameDelimited.h"

CFileNameDelimited::CFileNameDelimited()
{
    m_bDelimiter = '_';
}

bool CFileNameDelimited::append(LPCSTR pszFieldName, LPCSTR pszFieldValue)
{
    FILENAME_FIELD rField;

    rField.strName = pszFieldName;
    rField.strValue = pszFieldValue;

    push_back(rField);

    return true;
}

bool CFileNameDelimited::append(LPCSTR pszFieldName, long lFieldValue)
{
    char pszValue [20];
    ltoa(lFieldValue, pszValue, 10);
    return append(pszFieldName, pszValue);
}

bool CFileNameDelimited::append(LPCSTR pszFieldName, DATE dateValue, LPCSTR pszDateFormat)
{
    COleDateTime odt(dateValue);

    char * pszFormat = (char *) pszDateFormat;
    if (pszFormat == NULL)
        pszFormat = "%m%d%y";

    append(pszFieldName, (LPCSTR) odt.Format(pszFormat));

    return true;
}

int CFileNameDelimited::getIndex(LPCSTR pszFieldName)
{
    int nCount = size();

    for (int i = 0; i < nCount; i++)
    {
        if ((*this)[i].strName.compare(pszFieldName) == 0)
            return i;
    }

    return -1;
}

bool CFileNameDelimited::get(int nIdx, string & strValue)
{
    strValue = "";

    if (nIdx < 0 || nIdx >= size())
        return false;

    strValue = (*this)[nIdx].strValue;

    return true;
}

bool CFileNameDelimited::get(int nIdx, long & lFieldValue)
```

```
{
    bool fSuccess;

    string strValue;

    if (fSuccess = get(nIdx, strValue))
        lFieldValue = atol(strValue.c_str());
    else
        lFieldValue = 0;

    return fSuccess;
}

bool CFileNameDelimited::get(int nIdx, DATE & dateValue)
{
    bool fSuccess;

    string strValue;

    if (fSuccess = get(nIdx, strValue))
    {
        strValue.insert(4, "/");
        strValue.insert(2, "/");
        COleDateTime odt;
        odt.ParseDateTime(strValue.c_str());
        dateValue = (DATE) odt;
    }
    else
        dateValue = 0.0;

    return fSuccess;
}

bool CFileNameDelimited::set(int nIdx, LPCSTR pszValue)
{
    if (nIdx < 0)
        return false;

    // pad out vector up to occurrence referenced
    if (nIdx >= size())
    {
        for (int i = size(); i <= nIdx; i++)
            append("", "");
    }

    (*this)[nIdx].strValue = pszValue;

    return true;
}

bool CFileNameDelimited::set(int nIdx, long lFieldValue)
{
    char pszValue [20];
    ltoa(lFieldValue, pszValue, 10);
    return set(nIdx, pszValue);
}

bool CFileNameDelimited::set(int nIdx, DATE dateValue, LPCSTR pszDateFormat)
{
    COleDateTime odt(dateValue);

    char * pszFormat = (char *) pszDateFormat;
    if (pszFormat == NULL)
        pszFormat = "%m%d%y";

    return set(nIdx, (LPCSTR) odt.Format(pszDateFormat));
}
```



```
bool CFileNameDelimited::setFullName(LPCSTR pszFileName, bool fClear)
{
    string strValue;

    if (fClear)
        clear();

    string strName = pszFileName;

    // pick out the extension if it exist
    int nExtPos = strName.find_last_of('.');
    if (nExtPos != string::npos)
    {
        m_strExtension = strName.substr(nExtPos + 1, strName.size() - nExtPos);
        strName.resize(nExtPos);
    }

    int nIdx = 0;

    // parse the name out into fields and set them
    if (strName.size())
    {
        int nLastPos = 0;
        int nNewPos = 0;

        while ((nNewPos = strName.find(m_bDelimiter, nLastPos)) != string::npos)
        {
            strValue = strName.substr(nLastPos, nNewPos - nLastPos);
            set(nIdx++, strValue.c_str());
            nLastPos = nNewPos + 1;
        }

        strValue = strName.substr(nLastPos, nNewPos - nLastPos);
        set(nIdx++, strValue.c_str());
    }

    // if file name shorter than fields, clear values on fields
    int nSize = size();
    for (; nIdx < nSize; nIdx++)
        set(nIdx, "");

    return true;
}

void CFileNameDelimited::setExtension(LPCSTR pszExt)
{
    m_strExtension = pszExt;
}

bool CFileNameDelimited::getFullName(string & strFileName)
{
    strFileName = "";

    CFileNameDelimited::iterator it;
    for (it = begin(); it != end(); it++)
    {
        if (strFileName.size())
            strFileName += m_bDelimiter;

        strFileName += (*it).strValue;
    }

    if (m_strExtension.size())
    {
        strFileName += ".";
        strFileName += m_strExtension;
    }
}
```

```
    }

    return true;
}

////////////////////////////////////
// CFileNameKiosk

CFileNameKiosk::CFileNameKiosk()
{
    append("direction", "");
    append("kiosk_id", "");
    append("date", "");
}
```

```
#ifndef _filenameDelimited_h
#define _filenameDelimited_h

struct FILENAME_FIELD
{
    string      strName;
    string      strValue;

    FILENAME_FIELD & operator=(const FILENAME_FIELD & rField)
    {
        strName = rField.strName;
        strValue = rField.strValue;
        return *this;
    }
};

class CFileNameDelimited : public vector<FILENAME_FIELD>
{
protected:
    char          m_bDelimiter;
    string        m_strExtension;

public:
    CFileNameDelimited();
    bool append(LPCSTR pszFieldName, LPCSTR pszFieldValue);
    bool append(LPCSTR pszFieldName, long lFieldValue);
    bool append(LPCSTR pszFieldName, DATE dateValue, LPCSTR pszDateFormat = NULL);

    bool get(LPCSTR pszFieldName, string & strValue);
    bool get(LPCSTR pszFieldName, long & lFieldValue);
    bool get(LPCSTR pszFieldName, DATE & dateValue);

    bool set(LPCSTR pszFieldName, LPCSTR pszValue);
    bool set(LPCSTR pszFieldName, long lFieldValue);
    bool set(LPCSTR pszFieldName, DATE dateValue, LPCSTR pszDateFormat = NULL);

    bool get(int nIndex, string & strValue);
    bool get(int nIndex, long & lFieldValue);
    bool get(int nIndex, DATE & dateValue);

    bool set(int nIndex, LPCSTR pszValue);
    bool set(int nIndex, long lFieldValue);
    bool set(int nIndex, DATE dateValue, LPCSTR pszDateFormat = NULL);

    bool setFullName(LPCSTR pszFileName, bool fClear = false);
    void setExtension(LPCSTR pszExt);

    bool getFullName(string & strFileName);

    int getIndex(LPCSTR pszFieldName);
};

inline bool CFileNameDelimited::get(LPCSTR pszFieldName, string & strValue)
{
    return get(getIndex(pszFieldName), strValue);
}

inline bool CFileNameDelimited::get(LPCSTR pszFieldName, long & lFieldValue)
{
    return get(getIndex(pszFieldName), lFieldValue);
}

inline bool CFileNameDelimited::get(LPCSTR pszFieldName, DATE & dateValue)
{
    return get(getIndex(pszFieldName), dateValue);
}
```

```
inline bool CFileNameDelimited::set(LPCSTR pszFieldName, LPCSTR pszValue)
{
    return set(getIndex(pszFieldName), pszValue);
}
```

```
inline bool CFileNameDelimited::set(LPCSTR pszFieldName, long lFieldValue)
{
    return set(getIndex(pszFieldName), lFieldValue);
}
```

```
inline bool CFileNameDelimited::set(LPCSTR pszFieldName, DATE dateValue, LPCSTR
    pszDateFormat)
{
    return set(getIndex(pszFieldName), dateValue, pszDateFormat);
}
```

```
////////////////////////////////////
```

```
// CFileNameKiosk
```

```
class CFileNameKiosk : public CFileNameDelimited
```

```
{
public:
    CFileNameKiosk();
};
```

```
#endif
```

```
// LCKioskClient.cpp : Implementation of WinMain.
```

```
// Note: Proxy/Stub Information
```

```
//      To build a separate proxy/stub DLL,
```

```
//      run nmake -f LCKioskClientps.mk in the project directory.
```

```
#include "stdafx.h"
```

```
#include "resource.h"
```

```
#include <initguid.h>
```

```
#include "LCKioskClient.h"
```

```
#include "threadMain.h"
```

```
#include "registryKClient.h"
```

```
#include "LCKioskClient_i.c"
```

```
#include <stdio.h>
```

```
////////////////////////////////////
```

```
// MFC support
```

```
CKioskClientApp _theApp;
```

```
////////////////////////////////////
```

```
// ATL support
```

```
CServiceModule _Module;
```

```
////////////////////////////////////
```

```
//Global decalarations
```

```
CLogNTEvents    _logEvents("Kiosk Client");
```

```
CLogFile        _logFile("c:\\LCKioskClient.log");
```

```
CLogDebug       _logDebug;
```

```
CLogMulti       _logAll;
```

```
string          _strDefaultAlias;
```

```
BEGIN_OBJECT_MAP(ObjectMap)
```

```
END_OBJECT_MAP()
```

```
LPCTSTR FindOneOf(LPCTSTR p1, LPCTSTR p2)
```

```
{
    while (p1 != NULL && *p1 != NULL)
    {
        LPCTSTR p = p2;
        while (p != NULL && *p != NULL)
        {
            if (*p1 == *p)
                return CharNext(p1);
            p = CharNext(p);
        }
        p1 = CharNext(p1);
    }
    return NULL;
}
```

```
// Although some of these functions are big they are declared inline since they are only used once ✓
```

```
inline HRESULT CServiceModule::RegisterServer(BOOL bRegTypeLib, BOOL bService)
```

```
{
    HRESULT hr = CoInitialize(NULL);
    if (FAILED(hr))
        return hr;

    // Remove any previous service since it may point to
    // the incorrect file
    Uninstall();
}
```

```
// Add service entries
UpdateRegistryFromResource(IDR_LCKioskClient, TRUE);

// Adjust the AppID for Local Server or Service
CRegKey keyAppID;
LONG lRes = keyAppID.Open(HKEY_CLASSES_ROOT, _T("AppID"), KEY_WRITE);
if (lRes != ERROR_SUCCESS)
    return lRes;

CRegKey key;
lRes = key.Open(keyAppID, _T("{A8B06B0D-E231-11D3-B883-80F7BB000000}"), KEY_WRITE);
if (lRes != ERROR_SUCCESS)
    return lRes;
key.DeleteValue(_T("LocalService"));

if (bService)
{
    key.SetValue(_T("LCKioskClient"), _T("LocalService"));
    key.SetValue(_T("-Service"), _T("ServiceParameters"));
    // Create service
    Install();
}

// Add object entries
hr = CComModule::RegisterServer(bRegTypeLib);

CoUninitialize();
return hr;
}

inline HRESULT CServiceModule::UnregisterServer()
{
    HRESULT hr = CoInitialize(NULL);
    if (FAILED(hr))
        return hr;

    // Remove service entries
    UpdateRegistryFromResource(IDR_LCKioskClient, FALSE);
    // Remove service
    Uninstall();
    // Remove object entries
    CComModule::UnregisterServer(TRUE);
    CoUninitialize();
    return S_OK;
}

inline void CServiceModule::Init(_ATL_OBJMAP_ENTRY* p, HINSTANCE h, UINT nServiceNameID,
    const GUID* plibid)
{
    CComModule::Init(p, h, plibid);

    m_bService = TRUE;

    LoadString(h, nServiceNameID, m_szServiceName, sizeof(m_szServiceName) / sizeof
        (TCHAR));

    // set up the initial service status
    m_hServiceStatus = NULL;
    m_status.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
    m_status.dwCurrentState = SERVICE_STOPPED;
    m_status.dwControlsAccepted = SERVICE_ACCEPT_STOP;
    m_status.dwWin32ExitCode = 0;
    m_status.dwServiceSpecificExitCode = 0;
    m_status.dwCheckpoint = 0;
    m_status.dwWaitHint = 0;
}
```

```
LONG CServiceModule::Unlock()
{
    LONG l = CComModule::Unlock();
    if (l == 0 && !m_bService)
        PostThreadMessage(dwThreadId, WM_QUIT, 0, 0);
    return l;
}

BOOL CServiceModule::IsInstalled()
{
    BOOL bResult = FALSE;

    SC_HANDLE hSCM = ::OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);

    if (hSCM != NULL)
    {
        SC_HANDLE hService = ::OpenService(hSCM, m_szServiceName, SERVICE_QUERY_CONFIG);
        if (hService != NULL)
        {
            bResult = TRUE;
            ::CloseServiceHandle(hService);
        }
        ::CloseServiceHandle(hSCM);
    }
    return bResult;
}

inline BOOL CServiceModule::Install()
{
    if (IsInstalled())
        return TRUE;

    SC_HANDLE hSCM = ::OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
    if (hSCM == NULL)
    {
        MessageBox(NULL, _T("Couldn't open service manager"), m_szServiceName, MB_OK);
        return FALSE;
    }

    // Get the executable file path
    TCHAR szFilePath[_MAX_PATH];
    ::GetModuleFileName(NULL, szFilePath, _MAX_PATH);

    SC_HANDLE hService = ::CreateService(
        hSCM, m_szServiceName, m_szServiceName,
        SERVICE_ALL_ACCESS, SERVICE_WIN32_OWN_PROCESS,
        SERVICE_DEMAND_START, SERVICE_ERROR_NORMAL,
        szFilePath, NULL, NULL, _T("RPCSS\0"), NULL, NULL);

    if (hService == NULL)
    {
        ::CloseServiceHandle(hSCM);
        MessageBox(NULL, _T("Couldn't create service"), m_szServiceName, MB_OK);
        return FALSE;
    }

    ::CloseServiceHandle(hService);
    ::CloseServiceHandle(hSCM);
    return TRUE;
}

inline BOOL CServiceModule::Uninstall()
{
    if (!IsInstalled())
        return TRUE;

    SC_HANDLE hSCM = ::OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
```

```

if (hSCM == NULL)
{
    MessageBox(NULL, _T("Couldn't open service manager"), m_szServiceName, MB_OK);
    return FALSE;
}

SC_HANDLE hService = ::OpenService(hSCM, m_szServiceName, SERVICE_STOP | DELETE);

if (hService == NULL)
{
    ::CloseServiceHandle(hSCM);
    MessageBox(NULL, _T("Couldn't open service"), m_szServiceName, MB_OK);
    return FALSE;
}

SERVICE_STATUS status;
::ControlService(hService, SERVICE_CONTROL_STOP, &status);

BOOL bDelete = ::DeleteService(hService);
::CloseServiceHandle(hService);
::CloseServiceHandle(hSCM);

if (bDelete)
    return TRUE;

MessageBox(NULL, _T("Service could not be deleted"), m_szServiceName, MB_OK);
return FALSE;
}

```

```

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Logging functions
void CServiceModule::LogEvent(LPCTSTR pFormat, ...)

```

```
TCHAR chMsg[2048];
va_list pArg;

va_start(pArg, pFormat);
_vstprintf(chMsg, pFormat, pArg);
va_end(pArg);

CLogMsgEvent(LCEV_GENERIC, -1, chMsg).Post(_logAll);
}
```

```
//////////////////////////////////////✓  
    ///
```

```
// Service startup and registration
inline void CServiceModule::Start()
{
    SERVICE_TABLE_ENTRY st[] =
    {
        { m_szServiceName, _ServiceMain },
        { NULL, NULL }
    };
    if (m_bService && !::StartServiceCtrlDispatcher(st))
    {
        m_bService = FALSE;
    }
    if (m_bService == FALSE)
        Run();
}

```

```
inline void CServiceModule::ServiceMain(DWORD /* dwArgc */, LPTSTR* /* lpszArgv */)
{
    // Register the control request handler
    m_status.dwCurrentState = SERVICE_START_PENDING;
    m_hServiceStatus = RegisterServiceCtrlHandler(m_szServiceName, _Handler);
    if (m_hServiceStatus == NULL)

```



```
{
    CLogMsgEvent("Handler not installed").Post(_logAll);
    return;
}
SetServiceStatus(SERVICE_START_PENDING);

m_status.dwWin32ExitCode = S_OK;
m_status.dwCheckPoint = 0;
m_status.dwWaitHint = 0;

// When the Run function returns, the service has stopped.
Run();

SetServiceStatus(SERVICE_STOPPED);
}

inline void CServiceModule::Handler(DWORD dwOpcode)
{
    switch (dwOpcode)
    {
    case SERVICE_CONTROL_STOP:
        SetServiceStatus(SERVICE_STOP_PENDING);
        PostThreadMessage(dwThreadId, WM_QUIT, 0, 0);
        break;
    case SERVICE_CONTROL_PAUSE:
        break;
    case SERVICE_CONTROL_CONTINUE:
        break;
    case SERVICE_CONTROL_INTERROGATE:
        break;
    case SERVICE_CONTROL_SHUTDOWN:
        break;
    default:
        CLogMsgEvent("Bad service request").Post(_logAll);
    }
}

void WINAPI CServiceModule::_ServiceMain(DWORD dwArgc, LPTSTR* lpszArgv)
{
    _Module.ServiceMain(dwArgc, lpszArgv);
}

void WINAPI CServiceModule::_Handler(DWORD dwOpcode)
{
    _Module.Handler(dwOpcode);
}

void CServiceModule::SetServiceStatus(DWORD dwState)
{
    m_status.dwCurrentState = dwState;
    ::SetServiceStatus(m_hServiceStatus, &m_status);
}

void CServiceModule::Run()
{
    _Module.dwThreadId = GetCurrentThreadId();

    HRESULT hr = CoInitializeEx(NULL, COINIT_MULTITHREADED);
    if (FAILED(hr))
    {
        CLogMsgEvent msg(LCEV_GENERIC, SVRTY_ERROR);
        msg << "CoInitializeEx() failed. Error = [0x" << std::hex << hr << "]";
        msg.Post(_logAll);
        return;
    }

    _ASSERT(SUCCEEDED(hr));
}
```

```

// This provides a NULL DACL which will allow access to everyone.
CSecurityDescriptor sd;
sd.InitializeFromThreadToken();
hr = CoInitializeSecurity(sd, -1, NULL, NULL,
    RPC_C_AUTHN_LEVEL_PKT, RPC_C_IMP_LEVEL_IMPERSONATE, NULL, EOAC_NONE, NULL);
_ASSERTE(SUCCEEDED(hr));

hr = _Module.RegisterClassObjects(CLSCTX_LOCAL_SERVER | CLSCTX_REMOTE_SERVER,
    REGCLS_MULTIPLEUSE);
_ASSERTE(SUCCEEDED(hr));

////////////////////////////////////
// MFC support
if (_theApp.InitApplication() == FALSE)
{
    CLogMsgEvent msg(LCEV_GENERIC, SVRTY_ERROR);
    msg << "_theApp.InitApplication() failed";
    msg.Post(_logAll);
    return;
}

if (_theApp.InitInstance() == FALSE)
{
    CLogMsgEvent msg(LCEV_GENERIC, SVRTY_ERROR);
    msg << "_theApp.InitInstance() failed";
    msg.Post(_logAll);
    _theApp.ExitInstance();
    return;
}

// end MFC support
////////////////////////////////////

CLogMsgEvent("Service started").Post(_logAll);
if (m_bService)
    SetServiceStatus(SERVICE_RUNNING);

_theApp.Run();

_theApp.ExitInstance();

CLogMsgEvent("Service stopped").Post(_logAll);

_Module.RevokeClassObjects();

CoUninitialize();
}

////////////////////////////////////
//

extern "C" int WINAPI _tWinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int nShowCmd)
{
    _logAll.AddLog(&_logEvents);
    _logDebug.Enabled(false);
    _logAll.AddLog(&_logFile);

#ifdef _DEBUG
    _logEvents.EnableTranslation(true);
    _logDebug.Enabled(true);
    _logAll.AddLog(&_logDebug);
#endif

    lpCmdLine = GetCommandLine(); //this line necessary for ATL_MIN_CRT
    _Module.Init(ObjectMap, hInstance, IDS_SERVICENAME, &LIBID_LCKIOSKCLIENTLib);
    _Module.m_bService = TRUE;

```

```
TCHAR szTokens[] = _T("-/");

LPCTSTR lpszToken = FindOneOf(lpCmdLine, szTokens);
while (lpszToken != NULL)
{
    if (lstrcmpi(lpszToken, _T("UnregServer"))==0)
        return _Module.UnregisterServer();

    // Register as Local Server
    if (lstrcmpi(lpszToken, _T("RegServer"))==0)
        return _Module.RegisterServer(TRUE, FALSE);

    // Register as Service
    if (lstrcmpi(lpszToken, _T("Service"))==0)
        return _Module.RegisterServer(TRUE, TRUE);

    // Initialize Configuration Registry Entries
    if (lstrcmpi(lpszToken, _T("InitReg"))==0)
    {
        CRegistryKClient reg;
        reg.buildInitial();
        return 0;
    }

    lpszToken = FindOneOf(lpszToken, szTokens);
}

// Are we Service or Local Server
CRegKey keyAppID;
LONG lRes = keyAppID.Open(HKEY_CLASSES_ROOT, _T("AppID"), KEY_READ);
if (lRes != ERROR_SUCCESS)
{
    CLogMsgEvent msg(LCEV_GENERIC, SVRTY_ERROR);
    msg << "Unable to open \"AppID\" from HKEY_CLASSES_ROOT";
    msg.Post(_logAll);
    return lRes;
}

CRegKey key;
lRes = key.Open(keyAppID, _T("{A8B06B0D-E231-11D3-B883-80F7BB000000}"), KEY_READ);
if (lRes != ERROR_SUCCESS)
{
    CLogMsgEvent msg(LCEV_GENERIC, SVRTY_ERROR);
    msg << "Unable to open \"{A8B06B0D-E231-11D3-B883-80F7BB000000}\" from
HKEY_CLASSES_ROOT/AppID";
    msg.Post(_logAll);
    return lRes;
}

TCHAR szValue[_MAX_PATH];
DWORD dwLen = _MAX_PATH;
lRes = key.QueryValue(szValue, _T("LocalService"), &dwLen);

_Module.m_bService = FALSE;
if (lRes == ERROR_SUCCESS)
    _Module.m_bService = TRUE;

// init MFC support
ASSERT(hPrevInstance == NULL);

// AFX internal initialization
if (!AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nShowCmd))
    CLogMsgEvent(LCEV_GENERIC, SVRTY_ERROR, "afxWinInit failed.").Post(_logAll);
else
    _Module.Start();
```

```
    // When we get here, the service has been stopped
    return _Module.m_status.dwWin32ExitCode;
}
```

```
/* this ALWAYS GENERATED file contains the definitions for the interfaces */

/* File created by MIDL compiler version 5.01.0164 */
/* at Sun Feb 13 13:18:39 2000
*/
/* Compiler settings for D:\Dev\Lifeclinic\LCServices\LCKioskClient\LCKioskClient.idl:
    Oicf (OptLev=i2), W1, Zp8, env=Win32, ms_ext, c_ext
    error checks: allocation ref bounds_check enum stub_data
*/
//@@MIDL_FILE_HEADING(  )

/* verify that the <rpcndr.h> version is high enough to compile this file*/
#ifndef __REQUIRED_RPCNDR_H_VERSION__
#define __REQUIRED_RPCNDR_H_VERSION__ 440
#endif

#include "rpc.h"
#include "rpcndr.h"

#ifndef __LCKioskClient_h__
#define __LCKioskClient_h__

#ifdef __cplusplus
extern "C"{
#endif

/* Forward Declarations */

/* header files for imported files */
#include "oaidl.h"
#include "ocidl.h"

void __RPC_FAR * __RPC_USER MIDL_user_allocate(size_t);
void __RPC_USER MIDL_user_free( void __RPC_FAR * );

#ifndef __LCKIOSKCLIENTLib_LIBRARY_DEFINED__
#define __LCKIOSKCLIENTLib_LIBRARY_DEFINED__

/* library LCKIOSKCLIENTLib */
/* [helpstring][version][uuid] */

EXTERN_C const IID LIBID_LCKIOSKCLIENTLib;
#endif /* __LCKIOSKCLIENTLib_LIBRARY_DEFINED__ */

/* Additional Prototypes for ALL interfaces */

/* end of Additional Prototypes */

#ifdef __cplusplus
}
#endif

#endif
```

```
#include "stdafx.h"
#include "Logging.h"
#include "Registry.h"

////////////////////////////////////
////////////////////////////////////
// Log messages
////////////////////////////////////
////////////////////////////////////
CLogMsg::CLogMsg()
{
    m_pszText = NULL;
}

CLogMsg::CLogMsg(LPCSTR pszMessage)
{
    m_pszText = NULL;
    if (pszMessage != NULL)
        *this << pszMessage;
}

CLogMsg::CLogMsg(string & strMessage)
{
    m_pszText = NULL;
    *this << strMessage;
}

CLogMsg::~CLogMsg()
{
    ReleaseBuffers();
}

CLogMsg & CLogMsg::Format(LPCSTR pszFormat, ...)
{
    Clear();
    va_list pArgs;
    va_start(pArgs, pszFormat);
    TCHAR pszBuffer [1024];
    vsprintf(pszBuffer, pszFormat, pArgs);
    va_end(pArgs);
    *this << pszBuffer;
    return *this;
}

void CLogMsg::Post(CLogBase & log)
{
    log.Post(this);
    return;
}

long CLogMsg::Event()
{
    return 0;
}

long CLogMsg::Severity()
{
    return EVENTLOG_SUCCESS;
}

TCHAR ** CLogMsg::Arguments(long * plArgCount)
{
    *plArgCount = 1;
    Text();
    return &m_pszText;
}
```

```
TCHAR * CLogMsg::Text()
{
    ReleaseBuffers();
    *this << '\0';
    TCHAR * pszText = str();
    int nLen = pcount();
    m_pszText = new TCHAR [nLen + 1];
    _tcscpy(m_pszText, pszText);
    freeze(false);
    return m_pszText;
}

void CLogMsg::ReleaseBuffers()
{
    if (m_pszText != NULL)
    {
        delete [] m_pszText;
        m_pszText = NULL;
    }
    return;
}

void CLogMsg::Clear()
{
    ReleaseBuffers();
    seekp(0);
    return;
}

void CLogMsg::appendError(_com_error & e)
{
    string strError = (char *) e.Description();
    HRESULT hr = e.Error();
    *this << "COM Error = [" << strError << "]. hr = [" << std::hex << hr << "].";
    return;
}

void CLogMsg::appendError(HRESULT hr)
{
    *this << "hr = [" << std::hex << hr << std::dec << "].";
    return;
}

void CLogMsg::appendError(CLogMsg & em)
{
    appendError((std::stringstream &) em);
}

void CLogMsg::appendError(std::stringstream & strmError)
{
    strmError << '\0';
    *this << strmError.str();
    strmError.freeze(false);
}

void CLogMsg::setError(_com_error & e)
{
    clear();
    appendError(e);
}

void CLogMsg::setError(HRESULT hr)
{
    clear();
    appendError(hr);
}
```

```
void CLogMsg::setError(LPCSTR pszError)
{
    clear();
    *this << pszError;
}

void CLogMsg::setError(CLogMsg & em)
{
    clear();
    appendError(em);
}

void CLogMsg::getError(string & strError)
{
    *this << '\0';
    strError = str();
    freeze(false);
    return;
}

string CLogMsg::getError()
{
    string strError;
    *this << '\0';
    strError = str();
    freeze(false);
    return strError;
}

void CLogMsg::getError(std::stringstream & strmError)
{
    *this << '\0';
    strmError << str();
    freeze(false);
    return;
}

////////////////////////////////////

const char CLogMsgEvent::bArgSep = '\t';

CLogMsgEvent::CLogMsgEvent()
{
    Init();
}

CLogMsgEvent::CLogMsgEvent(LPCSTR pszMessage)
: CLogMsg(pszMessage)
{
    Init();
}

CLogMsgEvent::CLogMsgEvent(string & strMessage)
: CLogMsg(strMessage)
{
    Init();
}

CLogMsgEvent::CLogMsgEvent(long lEventID, long lSeverity, LPCSTR pszMessage)
: CLogMsg(pszMessage)
{
    Init();
    m_lEventID = lEventID;
    m_lSeverity = lSeverity;
}

CLogMsgEvent::CLogMsgEvent(long lEventID, long lSeverity, string & strMessage)
```



```
:CLogMsg(strMessage)
{
    Init();
    m_lEventID = lEventID;
    m_lSeverity = lSeverity;
}

CLogMsgEvent::CLogMsgEvent(long lEventID, long lSeverity, _com_error & e)
{
    USES_CONVERSION;

    Init();
    m_lEventID = lEventID;
    m_lSeverity = lSeverity;

    *this << "0x" << std::hex << e.Error() << std::dec << bArgSep;
    BSTR bstrDesc = e.Description();
    if (bstrDesc != NULL)
        *this << W2T(bstrDesc);
    else
        *this << " ";
}

CLogMsgEvent::CLogMsgEvent(long lEventID, long lSeverity, HRESULT hr)
{
    Init();
    m_lEventID = lEventID;
    m_lSeverity = lSeverity;
    *this << "0x" << std::hex << hr;
}

CLogMsgEvent::~CLogMsgEvent()
{
    ReleaseBuffers();
}

inline void CLogMsgEvent::Init()
{
    m_lEventID = 0;
    m_lSeverity = -1;
    m_wArgCount = 0;
    m_ppszArgs = NULL;
}

void CLogMsgEvent::SetEvent(long lEventID, long lSeverity, LPCSTR pszMessage)
{
    Clear();
    m_lEventID = lEventID;
    m_lSeverity = lSeverity;
    if (pszMessage != NULL)
        *this << pszMessage;
}

long CLogMsgEvent::Event()
{
    return m_lEventID;
}

long CLogMsgEvent::Severity()
{
    if (m_lSeverity == -1)
    {
        if ((m_lEventID & 0xC0000000L) == 0xC0000000L)
            return EVENTLOG_ERROR_TYPE;
        else if (m_lEventID & 0x80000000L)
            return EVENTLOG_WARNING_TYPE;
        else if (m_lEventID & 0x40000000L)
```

```

        return EVENTLOG_INFORMATION_TYPE;
    else
        return EVENTLOG_SUCCESS;
    }
    else
        return m_lSeverity;
}

TCHAR ** CLogMsgEvent::Arguments(long * plArgCount)
{
    ReleaseBuffers();

    // get temp buffer
    strstream strmTemp;
    *this << '\0';
    strmTemp << str();
    freeze(false);

    // make sure double nulled
    strmTemp << '\0' << '\0';

    TCHAR * pszText = strmTemp.str();
    if (*pszText)
        m_wArgCount++;

    // make array of strings
    for (int i = 0; pszText[i]; i++)
    {
        if (pszText[i] == CLogMsgEvent::bArgSep)
        {
            pszText[i] = 0;
            m_wArgCount++;
        }
    }

    // if data, allocate arg array
    if (m_wArgCount)
    {
        int nLen = 0;
        m_ppszArgs = new TCHAR * [m_wArgCount];
        for (int i = 0; i < m_wArgCount; i++)
        {
            nLen = _tcslen(pszText);
            m_ppszArgs[i] = new TCHAR [nLen + 1];
            _tcsncpy(m_ppszArgs[i], pszText);
            pszText += nLen + 1;
        }
    }

    strmTemp.freeze(false);

    // return buffer
    *plArgCount = m_wArgCount;
    return m_ppszArgs;
}

TCHAR * CLogMsgEvent::Text()
{
    CLogMsg::ReleaseBuffers();

    // format message into temporary strstream
    *this << '\0';
    std::strstream strmTemp;
    strmTemp << "Event:0x" << std::hex << m_lEventID << ", Severity:" << std::dec <<
    Severity() << ", Text:";

    // if translation is turned on, then get message from message source

```

```

DWORD dwCharsReturned = 0;
if (CLogNTEvents::m_hMsgSrc != NULL)
{
    TCHAR pszBuff [2048];
    dwCharsReturned = FormatMessage(FORMAT_MESSAGE_FROM_HMODULE |
    FORMAT_MESSAGE_ARGUMENT_ARRAY,
    CLogNTEvents::m_hMsgSrc,
    m_lEventID,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
    pszBuff,
    2048,
    m_ppszArgs);

    if (dwCharsReturned)
    {
        // chop off line feed
        pszBuff[--dwCharsReturned] = 0;
        // move data to formatted message
        if (dwCharsReturned)
            strmTemp << pszBuff << '\0';
    }
}

// if translation not turned on or translation didn't work then put out argument data
if (!dwCharsReturned)
{
    strmTemp << str() << '\0';
    freeze(false);
}

// move temp stringstream into m_pszText and return pointer to m_pszText
int nLength = strmTemp.pcount();
m_pszText = new TCHAR [nLength + 1];
_tcsncpy(m_pszText, strmTemp.str(), nLength);
strmTemp.freeze(false);
m_pszText[nLength] = 0;

return m_pszText;
}

void CLogMsgEvent::ReleaseBuffers()
{
    CLogMsg::ReleaseBuffers();

    if (m_ppszArgs != NULL)
    {
        for (int i = 0; i < m_wArgCount; i++)
            delete [] m_ppszArgs[i];
        delete [] m_ppszArgs;
        m_ppszArgs = NULL;
        m_wArgCount = 0;
    }

    return;
}

////////////////////////////////////
////////////////////////////////////
// Logs
////////////////////////////////////
////////////////////////////////////

CLogBase::CLogBase()
{
    m_fEnabled = true;
    m_nIndent = 0;
}

```

```
CLogBase::CLogBase(LPCSTR pszResourceName)
{
    m_fEnabled = true;
    m_strResourceName = pszResourceName;
    m_nIndent = 0;
}

void CLogBase::ResourceName(LPCSTR pszResourceName)
{
    m_strResourceName = pszResourceName;
    return;
}

void CLogBase::Post(CLogMsg * pmsgLog)
{
    return;
}

void CLogBase::Open()
{
    return;
}

void CLogBase::Close()
{
    return;
}

////////////////////////////////////
HINSTANCE CLogNTEvents::m_hMsgSrc = NULL;

CLogNTEvents::CLogNTEvents()
: CLogBase()
{
}

CLogNTEvents::CLogNTEvents(LPCSTR pszResourceName)
: CLogBase(pszResourceName)
{
}

void CLogNTEvents::Post(CLogMsg * pmsgLog)
{
    if (!m_fEnabled)
        return;

    HANDLE hEventSource = RegisterEventSource(NULL, m_strResourceName.c_str());
    if (hEventSource != NULL)
    {
        long lArgCount;
        TCHAR ** pszArgs = pmsgLog->Arguments(&lArgCount);
        ReportEvent(hEventSource, pmsgLog->Severity(), 0, pmsgLog->Event(), NULL,
lArgCount,
        0, (const TCHAR **) pszArgs, NULL);
        DeregisterEventSource(hEventSource);
    }
}

void CLogNTEvents::EnableTranslation(bool fEnable)
{
    if (fEnable)
    {
        if (!m_hMsgSrc)
            m_hMsgSrc = LoadMessageSource();
    }
    else
}
```

```
{
    if (m_hMsgSrc)
    {
        FreeLibrary(m_hMsgSrc);
        m_hMsgSrc = NULL;
    }
}

return;
}

HINSTANCE CLogNTEvents::LoadMessageSource()
{
    CRegistry    regLocal;

    // get the name of the resource
    if (!regLocal.Connect(CRegistry::keyLocalMachine))
        return NULL;

    string strKey("SYSTEM\\CurrentControlSet\\Services\\EventLog\\Application\\");
    strKey += m_strResourceName;
    if (!regLocal.Open(strKey.c_str()))
        return NULL;

    string strDLL;
    if (!regLocal.GetValue("EventMessageFile", strDLL))
        return NULL;

    // load the library
    return LoadLibrary(strDLL.c_str());
}

////////////////////////////////////

CLogFile::CLogFile()
: CLogBase()
{
}

CLogFile::CLogFile(LPCSTR pszResourceName)
: CLogBase(pszResourceName)
{
    m_streamIO.open(pszResourceName, ios_base::out | ios_base::trunc);
}

void CLogFile::Open(LPCSTR pszFileName)
{
    Close();
    m_strResourceName = pszFileName;
    Open();
}

void CLogFile::Open()
{
    if (!m_streamIO.is_open())
        m_streamIO.open(m_strResourceName.c_str(), ios_base::out | ios_base::trunc);
}

void CLogFile::Close()
{
    if (m_streamIO.is_open())
        m_streamIO.close();
    return;
}

void CLogFile::Post(CLogMsg * pmsgLog)
```

```

{
    if (!m_fEnabled)
        return;

    Lock();
    if (m_streamIO.is_open())
    {
        string strTabs(m_nIndent, '\t');
        m_streamIO << strTabs << pmsgLog->Text() << '\n';
        m_streamIO.flush();
    }
    Unlock();

    return;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

CLogDebug::CLogDebug()
{
}

void CLogDebug::Post(CLogMsg * pmsgLog)
{
    if (!m_fEnabled)
        return;
    if (m_nIndent)
    {
        string strTabs(m_nIndent, '\t');
        OutputDebugString(strTabs.c_str());
    }
    OutputDebugString(pmsgLog->Text());
    OutputDebugString("\n");
    return;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

CLogMulti::CLogMulti()
{
}

void CLogMulti::AddLog(CLogBase * plog)
{
    Lock();
    m_collLogs.push_back(plog);
    Unlock();
}

void CLogMulti::RemoveLog(CLogBase * plog)
{
    Lock();
    if (plog != NULL)
        m_collLogs.remove(plog);
    else
        m_collLogs.erase(m_collLogs.begin(), m_collLogs.end());
    Unlock();
    return;
}

void CLogMulti::Post(CLogMsg * pmsgLog)
{
    if (!m_fEnabled)
        return;

    Lock();
    list<CLogBase *>::iterator    itLogs;

```

```
    for (itLogs = m_collLogs.begin(); itLogs != m_collLogs.end(); itLogs++)
        pmsgLog->Post(*itLogs);
    Unlock();

    return;
}

void CLogMulti::Open()
{
    Lock();
    list<CLogBase *>::iterator itLogs;
    for (itLogs = m_collLogs.begin(); itLogs != m_collLogs.end(); itLogs++)
        (*itLogs)->Open();
    Unlock();
}

void CLogMulti::Close()
{
    Lock();
    list<CLogBase *>::iterator itLogs;
    for (itLogs = m_collLogs.begin(); itLogs != m_collLogs.end(); itLogs++)
        (*itLogs)->Close();
    Unlock();
}

string CTimeStamp::LocalTime()
{
    SYSTEMTIME tm;
    GetLocalTime(&tm);
    char pBuff [30];
    sprintf(pBuff, "%02d:%02d:%02d.%d", tm.wHour, tm.wMinute, tm.wSecond, tm.
        wMilliseconds);
    return string(pBuff);
}
```

✓


```

{
protected:
    fstream          m_streamIO;

public:
    CLogFile();
    CLogFile(LPCSTR pszResourceName);
    void Open(LPCSTR pszFileName);
    virtual void Post(CLogMsg * pmsgLog);
    virtual void Open();
    virtual void Close();
};

////////////////////////////////////////////////////////////////

class CLogDebug : public CLogBase
{
public:
    CLogDebug();
    virtual void Post(CLogMsg * pmsgLog);
};

////////////////////////////////////////////////////////////////

class CLogMulti : public CLogBase
{
protected:
    list<CLogBase *>    m_collLogs;

public:
    CLogMulti();
    void AddLog(CLogBase * plog);
    void RemoveLog(CLogBase * plog);
    virtual void Post(CLogMsg * pmsgLog);
    virtual void Open();
    virtual void Close();
};

////////////////////////////////////////////////////////////////
// message objects
////////////////////////////////////////////////////////////////

class CLogMsg : public std::stringstream
{
protected:
    TCHAR *          m_pszText;
    virtual void ReleaseBuffers();

public:
    CLogMsg();
    CLogMsg(LPCSTR pszMessage);
    CLogMsg(string & strMessage);
    virtual ~CLogMsg();
    CLogMsg & Format(LPCSTR pszFormat, ...);
    virtual void Post(CLogBase & log);
    virtual long Event();
    virtual long Severity();
    virtual TCHAR ** Arguments(long * plArgCount);
    virtual TCHAR * Text();
    virtual void Clear();
    void appendError(_com_error & e);
    void appendError(HRESULT hr);
    void appendError(CLogMsg & em);
    void appendError(std::stringstream & strmError);
    void setError(_com_error & e);
};

```

```

    void setError(HRESULT hr);
    void setError(LPCSTR pszError);
    void setError(CLogMsg & em);
    void getError(string & strError);
    string getError();
    void getError(std::stringstream & strmError);
    void clear(){Clear();}
};

////////////////////////////////////

enum { SVRTY_DEFAULT    = -1,
        SVRTY_SUCCESS  = EVENTLOG_SUCCESS,
        SVRTY_ERROR    = EVENTLOG_ERROR_TYPE,
        SVRTY_WARNING  = EVENTLOG_WARNING_TYPE,
        SVRTY_INFO     = EVENTLOG_INFORMATION_TYPE };

class CLogMsgEvent : public CLogMsg
{
protected:
    long          m_lEventID;
    long          m_lSeverity;
    short         m_wArgCount;
    TCHAR **      m_ppszArgs;

    void Init();
    virtual void ReleaseBuffers();

public:
    static const char    bArgSep;

public:
    CLogMsgEvent();
    CLogMsgEvent(LPCSTR pszMessage);
    CLogMsgEvent(string & strMessage);
    CLogMsgEvent(long lEventID, long lSeverity = -1, LPCSTR pszMessage = NULL);
    CLogMsgEvent(long lEventID, long lSeverity, string & strMessage);
    CLogMsgEvent(long lEventID, long lSeverity, _com_error & e);
    CLogMsgEvent(long lEventID, long lSeverity, HRESULT hr);
    ~CLogMsgEvent();
    void SetEvent(long lEventID, long lSeverity = -1, LPCSTR pszMessage = NULL);
    virtual long Event();
    virtual long Severity();
    virtual TCHAR ** Arguments(long * plArgCount);
    virtual TCHAR * Text();
};

class CTimeStamp
{
public:
    static string LocalTime();
};

#endif

```



```

    {
        _recursively_delete_all_sub_keys( child_key_handle, temporary_key_name );

        return_value = RegEnumKey( child_key_handle, 0, temporary_key_name, MAX_PATH );
    }

    delete [] temporary_key_name;
    temporary_key_name = NULL;
    RegCloseKey( child_key_handle );
    return_value = RegDeleteKey( key_handle, key_name );
    return( return_value );
}

/*****
Function name: CRegistry::CRegistry
Description   :
Return type  :
*****/
CRegistry::CRegistry()
{
    m_Initialize();
}

/*****
Function name: CRegistry::~~CRegistry
Description   :
Return type  :
*****/
CRegistry::~~CRegistry()
{
    if ( m_RegistryHandle != (HKEY) NULL )
    {
        Close();
    }

    m_Initialize();
}

/*****
Function name: CRegistry::m_Initialize
Description   :
Return type  : void
Argument     : void
*****/
void CRegistry::m_Initialize( void )
{
    _ASSERT( this );

    /*
    ** Make sure everything is zeroed out
    */

    m_ClassName.erase();
    m_ComputerName.erase();
    m_KeyName.erase();
    m_RegistryName.erase();

    m_KeyHandle          = (HKEY) NULL;
    m_ErrorCode          = 0L;
    m_NumberOfSubkeys    = 0;
    m_LongestSubkeyNameLength = 0;
    m_LongestClassNameLength = 0;
    m_NumberOfValues      = 0;
    m_LongestValueNameLength = 0;
    m_LongestValueDataLength = 0;
    m_SecurityDescriptorLength = 0;
    m_LastWriteTime.dwLowDateTime = 0;

```

```

    m_LastWriteTime.dwHighDateTime = 0;
    m_RegistryHandle = (HKEY) NULL;
}

/*****
Function name: CRegistry::Close
Description   :
Return type  : BOOL
Argument     : void
*****/
BOOL CRegistry::Close( void )
{
    _ASSERTE( this );

    if ( m_KeyHandle != (HKEY) NULL )
    {
        ::RegCloseKey( m_KeyHandle );
        m_KeyHandle = (HKEY) NULL;
    }

    if ( m_RegistryHandle == (HKEY) NULL )
    {
        return( TRUE );
    }

    m_ErrorCode = ::RegCloseKey( m_RegistryHandle );

    if ( m_ErrorCode == ERROR_SUCCESS )
    {
        m_RegistryHandle = (HKEY) NULL;
        m_Initialize();

        return( TRUE );
    }
    else
    {
        return( FALSE );
    }
}

/*****
Function name: CRegistry::Connect
Description   :
Return type  : BOOL
Argument     : HKEY key_to_open
Argument     : LPCTSTR name_of_computer
*****/
BOOL CRegistry::Connect( const _Keys key_to_open, LPCTSTR name_of_computer )
{
    _ASSERTE( this );

    // We were passed a pointer, do not trust it
    try
    {
        /*
        ** name_of_computer can be NULL
        */

        if ( key_to_open == keyClassesRoot || key_to_open == keyCurrentUser )
        {
            if ( name_of_computer == NULL )
            {
                m_RegistryHandle = (HKEY)key_to_open;
                m_ErrorCode = ERROR_SUCCESS;
            }
            else
            {

```

```
    /*  
    ** NT won't allow you to connect to these hives via RegConnectRegistry so we  
    'll just skip that step  
    */  
    m_ErrorCode = ERROR_INVALID_HANDLE;  
    }  
    }  
    else  
    {  
        // Thanks to Paul Ostrowski [postrowski@xantel.com] for finding UNICODE bug here  
        // RegConnectRegistry is not const correct  
        m_ErrorCode = ::RegConnectRegistry( (LPTSTR) name_of_computer, (HKEY)key_to_open, &m_RegistryHandle );  
    }  
  
    if ( m_ErrorCode == ERROR_SUCCESS )  
    {  
        if ( name_of_computer == NULL )  
        {  
            TCHAR computer_name[ MAX_PATH ];  
            DWORD size = MAX_PATH;  
  
            if ( ::GetComputerName( computer_name, &size ) == FALSE )  
            {  
                m_ComputerName.erase();  
            }  
            else  
            {  
                m_ComputerName = computer_name;  
            }  
        }  
        else  
        {  
            m_ComputerName = name_of_computer;  
        }  
  
        //  
        // It would be nice to use a switch statement here but I get a "not integral" error!  
        //  
        if ( (HKEY)key_to_open == HKEY_LOCAL_MACHINE )  
        {  
            m_RegistryName = TEXT( "HKEY_LOCAL_MACHINE" );  
        }  
        else if ( (HKEY)key_to_open == HKEY_CLASSES_ROOT )  
        {  
            m_RegistryName = TEXT( "HKEY_CLASSES_ROOT" );  
        }  
        else if ( (HKEY)key_to_open == HKEY_USERS )  
        {  
            m_RegistryName = TEXT( "HKEY_USERS" );  
        }  
        else if ( (HKEY)key_to_open == HKEY_CURRENT_USER )  
        {  
            m_RegistryName = TEXT( "HKEY_CURRENT_USER" );  
        }  
        else if ( (HKEY)key_to_open == HKEY_PERFORMANCE_DATA )  
        {  
            m_RegistryName = TEXT( "HKEY_PERFORMANCE_DATA" );  
        }  
        #if ( WINVER >= 0x400 )  
        else if ( (HKEY)key_to_open == HKEY_CURRENT_CONFIG )  
        {  
            m_RegistryName = TEXT( "HKEY_CURRENT_CONFIG" );  
        }  
        else if ( (HKEY)key_to_open == HKEY_DYN_DATA )  
        {  
            m_RegistryName = TEXT( "HKEY_DYN_DATA" );  
        }  
    }  
}
```

```

    {
        m_RegistryName = TEXT( "HKEY_DYN_DATA" );
    }
#endif
    else
    {
        m_RegistryName = TEXT( "Unknown" );
    }

    return( TRUE );
}
else
{
    return( FALSE );
}
}
catch( ... )
{
    m_ErrorCode = ERROR_EXCEPTION_IN_SERVICE;
    return( FALSE );
}
}

/*****
Function name: CRegistry::Create
Description   :
Return type  : BOOL
Argument     : LPCTSTR          name_of_subkey
Argument     : LPCTSTR          name_of_class
Argument     : CreateOptions    options
Argument     : CreatePermissions permissions
Argument     : LPSECURITY_ATTRIBUTES security_attributes_p
Argument     : CreationDisposition * disposition_p
*****/
BOOL CRegistry::Create( LPCTSTR          name_of_subkey,
                       LPCTSTR          name_of_class,
                       CreateOptions    options,
                       CreatePermissions permissions,
                       LPSECURITY_ATTRIBUTES security_attributes_p,
                       CreationDisposition * disposition_p )
{
    _ASSERT( this );
    _ASSERT( name_of_subkey != NULL );

    if ( name_of_subkey == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    // We were passed a pointer, do not trust it
    try
    {
        DWORD disposition = 0;

        if ( name_of_class == NULL )
        {
            name_of_class = TEXT( "" ); // Paul Ostrowski [postrowski@xantel.com]
        }

        if ( m_KeyHandle != (HKEY) NULL )
        {
            ::RegCloseKey( m_KeyHandle );
            m_KeyHandle = (HKEY) NULL;
        }

        m_ErrorCode = ::RegCreateKeyEx( m_RegistryHandle,

```

```

        name_of_subkey,
        (DWORD) 0,
        (LPTSTR) name_of_class, // Paul Ostrowski
[postrowski@zantel.com]
        options,
        permissions,
        security_attributes_p,
        &m_KeyHandle,
        &disposition );

    if ( m_ErrorCode == ERROR_SUCCESS )
    {
        if ( disposition_p != NULL )
        {
            *disposition_p = (CreationDisposition) disposition;
        }

        m_KeyName = name_of_subkey;

        return( TRUE );
    }
    else
    {
        return( FALSE );
    }
}
catch( ... )
{
    m_ErrorCode = ERROR_EXCEPTION_IN_SERVICE;
    return( FALSE );
}
}

/*****
Function name: CRegistry::DeleteKey
Description   :
Return type   : BOOL
Argument      : LPCTSTR name_of_key_to_delete
*****/
BOOL CRegistry::DeleteKey( LPCTSTR name_of_key_to_delete )
{
    _ASSERT( this );
    _ASSERT( name_of_key_to_delete != NULL );

    if ( name_of_key_to_delete == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    // We were passed a pointer, do not trust it
    try
    {
        /*
        ** You can't delete a key given a full path. What you have to do is back up one
        level and then do a delete
        */

        string full_key_name = name_of_key_to_delete;

        if ( full_key_name.find( TEXT( '\\\\' ) ) == (-1) )
        {
            /*
            ** User had not given us a full path so assume the name of the key he passed us
            ** is a key off of the current key
            */

```



```

        m_ErrorCode = ::_recursively_delete_all_sub_keys( m_KeyHandle,
name_of_key_to_delete );
    }
    else
    {
        int last_back_slash_location = full_key_name.size() - 1;

        /*
        ** We know this loop will succeed because a back slash was found in the above if
statement
        */

        while( full_key_name[ last_back_slash_location ] != TEXT( '\\' ) )
        {
            last_back_slash_location--;
        }

        string currently_opened_key_name = m_KeyName;

        string parent_key_name = full_key_name.substr( 0, last_back_slash_location );
        int nCount = ( full_key_name.size() - last_back_slash_location ) - 1;
        string child_key_name = full_key_name.substr( full_key_name.size() - nCount,
nCount);

        /*
        ** Now we open the parent key and delete the child
        */

        if ( Open( parent_key_name.c_str() ) != FALSE )
        {
            m_ErrorCode = ::_recursively_delete_all_sub_keys( m_KeyHandle, child_key_name.
c_str() );
        }
        else
        {
            {
                m_KeyName = currently_opened_key_name;
                return( FALSE );
            }
        }

        if ( m_ErrorCode == ERROR_SUCCESS )
        {
            return( TRUE );
        }
        else
        {
            return( FALSE );
        }
    }
}
catch( ... )
{
    m_ErrorCode = ERROR_EXCEPTION_IN_SERVICE;
    return( FALSE );
}
}

/*
*****
Function name: CRegistry::DeleteValue
Description   :
Return type   : BOOL
Argument      : LPCTSTR name_of_value_to_delete
*****
BOOL CRegistry::DeleteValue( LPCTSTR name_of_value_to_delete )
{
    _ASSERT( this );

    /*

```

```

    /* name_of_value_to_delete can be NULL
    */

    // We were passed a pointer, do not trust it
    try
    {
        m_ErrorCode = ::RegDeleteValue( m_KeyHandle, (LPTSTR) name_of_value_to_delete );

        if ( m_ErrorCode == ERROR_SUCCESS )
        {
            return( TRUE );
        }
        else
        {
            return( FALSE );
        }
    }
    catch( ... )
    {
        m_ErrorCode = ERROR_EXCEPTION_IN_SERVICE;
        return( FALSE );
    }
}

```

```

/*****

```

```

Function name: CRegistry::EnumerateKeys
Description   :
Return type   : BOOL
Argument      : const DWORD subkey_index
Argument      : string& subkey_name
Argument      : string& class_name

```

```

*****/

```

```

BOOL CRegistry::EnumerateKeys( const DWORD subkey_index, string& subkey_name, string&
    class_name )

```

```

{
    _ASSERT( this );

    TCHAR subkey_name_string[ 2048 ];
    TCHAR class_name_string[ 2048 ];

    DWORD size_of_subkey_name_string = (sizeof(subkey_name_string)/sizeof(*
        (subkey_name_string))) - 1;
    DWORD size_of_class_name_string = (sizeof(class_name_string)/sizeof(*
        (class_name_string))) - 1;

    ::ZeroMemory( subkey_name_string, sizeof( subkey_name_string ) );
    ::ZeroMemory( class_name_string, sizeof( class_name_string ) );

    m_ErrorCode = ::RegEnumKeyEx( m_KeyHandle,
        subkey_index,
        subkey_name_string,
        &size_of_subkey_name_string,
        NULL,
        class_name_string,
        &size_of_class_name_string,
        &m_LastWriteTime );

    if ( m_ErrorCode == ERROR_SUCCESS )
    {
        subkey_name = subkey_name_string;
        class_name = class_name_string;
        return( TRUE );
    }
    else
    {
        return( FALSE );
    }
}

```

```

)

/* *****
Function name: CRegistry::EnumerateValues
Description   :
Return type   : BOOL
Argument      : const DWORD    value_index
Argument      : string&        name_of_value
Argument      : KeyValueTypes& type_code
Argument      : LPBYTE         data_buffer
Argument      : DWORD&         size_of_data_buffer
***** */
BOOL CRegistry::EnumerateValues( const DWORD    value_index,
                                string&        name_of_value,
                                KeyValueTypes& type_code,
                                LPBYTE         data_buffer,
                                DWORD&         size_of_data_buffer )
{
    _ASSERT( this );

    /*
    ** data_buffer and size_of_data_buffer can be NULL
    */

    DWORD temp_type_code = type_code;

    TCHAR temp_name[ 2048 ];

    ::ZeroMemory( temp_name, sizeof( temp_name ) );
    DWORD temp_name_size = (sizeof(temp_name)/sizeof(*(temp_name)));

    // We were passed a pointer, do not trust it
    try
    {
        m_ErrorCode = ::RegEnumValue( m_KeyHandle,
                                      value_index,
                                      temp_name,
                                      &temp_name_size,
                                      NULL,
                                      &temp_type_code,
                                      data_buffer,
                                      &size_of_data_buffer );

        if ( m_ErrorCode == ERROR_SUCCESS )
        {
            type_code    = (KeyValueTypes) temp_type_code;
            name_of_value = temp_name;

            return( TRUE );
        }
        else
        {
            return( FALSE );
        }
    }
    catch( ... )
    {
        m_ErrorCode = ERROR_EXCEPTION_IN_SERVICE;
        return( FALSE );
    }
}

/* *****
Function name: CRegistry::Flush
Description   :
Return type   : BOOL
Argument      : void
***** */

```

```

*****
BOOL CRegistry::Flush( void )
{
    _ASSERT( this );

    m_ErrorCode = ::RegFlushKey( m_KeyHandle );

    if ( m_ErrorCode == ERROR_SUCCESS )
    {
        return( TRUE );
    }
    else
    {
        return( FALSE );
    }
}

/*****
Function name: CRegistry::GetBinaryValue
Description   :
Return type   : BOOL
Argument      : LPCTSTR name_of_value
Argument      : BYTE return_array[]
Argument      : DWORD& num_bytes_read
*****/
BOOL CRegistry::GetBinaryValue( LPCTSTR name_of_value, BYTE return_array[], DWORD&
    num_bytes_read )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    // Thanks go to Chris Hines (ChrisHines@msn.com) for finding
    // a bug here. If you add entries to the key, then the
    // information retrieved via QueryInfo() may be invalid. This
    // will screw you here. So, we must make sure our information
    // is correct before we attempt to *use* the data.

    QueryInfo();
    DWORD size_of_buffer = m_LongestValueDataLength;
    LPBYTE memory_buffer = (LPBYTE) new BYTE[ size_of_buffer ];

    if ( memory_buffer == NULL )
    {
        m_ErrorCode = ::GetLastError();
        return( FALSE );
    }

    BOOL return_value = TRUE;
    KeyValueTypes type = typeBinary;

    if ( QueryValue( name_of_value, type, memory_buffer, size_of_buffer ) != FALSE )
    {
        DWORD index = 0;

        while( index < size_of_buffer )
        {
            return_array[index] = memory_buffer[index];
            index++;
        }

        num_bytes_read = size_of_buffer;
    }
}

```

```

        return_value = TRUE;
    }
    else
    {
        return_value = FALSE;
    }

    delete [] memory_buffer;
    return( return_value );
}

/*****
Function name: CRegistry::GetClassName
Description   :
Return type   : void
Argument      : string& class_name
*****/
void CRegistry::GetClassName( string& class_name ) const
{
    class_name = m_ClassName;
}

/*****
Function name: CRegistry::GetComputerName
Description   :
Return type   : void
Argument      : string& computer_name
*****/
void CRegistry::GetComputerName( string& computer_name ) const
{
    computer_name = m_ComputerName;
}

/*****
Function name: CRegistry::GetDoubleWordValue
Description   :
Return type   : BOOL
Argument      : LPCTSTR name_of_value
Argument      : DWORD& return_value
*****/
BOOL CRegistry::GetDoubleWordValue( LPCTSTR name_of_value, DWORD& return_value )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    DWORD size_of_buffer = sizeof( DWORD );
    KeyValueTypes type = typeDoubleWord;
    return( QueryValue( name_of_value, type, (LPBYTE) &return_value, size_of_buffer ) );
}

/*****
Function name: CRegistry::GetErrorCode
Description   :
Return type   : BOOL
Argument      : void
*****/
BOOL CRegistry::GetErrorCode( void ) const
{
    _ASSERT( this );
    return( m_ErrorCode );
}

```

```

/*****
Function name: CRegistry::GetKeyName
Description   :
Return type  : void
Argument     : string& key_name
*****/
void CRegistry::GetKeyName( string& key_name ) const
{
    key_name = m_KeyName;
}

/*****
Function name: CRegistry::GetNumberOfSubkeys
Description   :
Return type  : DWORD
Argument     : void
*****/
DWORD CRegistry::GetNumberOfSubkeys( void ) const
{
    return( m_NumberOfSubkeys );
}

/*****
Function name: CRegistry::GetNumberOfValues
Description   :
Return type  : DWORD
Argument     : void
*****/
DWORD CRegistry::GetNumberOfValues( void ) const
{
    return( m_NumberOfValues );
}

/*****
Function name: CRegistry::GetRegistryName
Description   :
Return type  : void
Argument     : string& registry_name
*****/
void CRegistry::GetRegistryName( string& registry_name ) const
{
    registry_name = m_RegistryName;
}

/*****
Function name: CRegistry::GetStringValue
Description   :
Return type  : BOOL
Argument     : LPCTSTR name_of_value
Argument     : string& return_string
*****/
BOOL CRegistry::GetStringValue( LPCTSTR name_of_value, string& return_string )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    TCHAR temp_string[ 2048 ];
    DWORD size_of_buffer = 2048;

    ::ZeroMemory( temp_string, sizeof( temp_string ) );

```

```

    KeyValueTypes type = typeString;

    if ( QueryValue( name_of_value, type, (LPBYTE) temp_string, size_of_buffer ) != FALSE )
    {
        return_string = temp_string;
        return( TRUE );
    }
    else
    {
        return_string.erase();
        return( FALSE );
    }
}

/* *****
Function name: CRegistry::GetValue
Description   :
Return type   : BOOL
Argument      : LPCTSTR name_of_value
Argument      : DWORD& return_value
***** */
BOOL CRegistry::GetValue( LPCTSTR name_of_value, DWORD& return_value )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    return( GetDoubleWordValue( name_of_value, return_value ) );
}

/* *****
Function name: CRegistry::GetValue
Description   :
Return type   : BOOL
Argument      : LPCTSTR name_of_value
Argument      : string& return_string
***** */
BOOL CRegistry::GetValue( LPCTSTR name_of_value, string& return_string )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    return( GetStringValue( name_of_value, return_string ) );
}

/* *****
Function name: CRegistry::Open
Description   :
Return type   : BOOL
Argument      : LPCTSTR name_of_subkey_to_open
Argument      : const CreatePermissions security_access_mask
***** */
BOOL CRegistry::Open( LPCTSTR name_of_subkey_to_open, const CreatePermissions
    security_access_mask )
{

```

```

    _ASSERTE( this );

    /*
    ** name_of_subkey_to_open can be NULL
    */

    // We were passed a pointer, do not trust it
    try
    {
        if ( m_KeyHandle != (HKEY) NULL )
        {
            ::RegCloseKey( m_KeyHandle );
            m_KeyHandle = (HKEY) NULL;
        }

        m_ErrorCode = ::RegOpenKeyEx( m_RegistryHandle, name_of_subkey_to_open, NULL,
        security_access_mask, &m_KeyHandle );

        if ( m_ErrorCode == ERROR_SUCCESS )
        {
            QueryInfo();
            m_KeyName = name_of_subkey_to_open;

            return( TRUE );
        }
        else
        {
            return( FALSE );
        }
    }
    catch( ... )
    {
        m_ErrorCode = ERROR_EXCEPTION_IN_SERVICE;
        return( FALSE );
    }
}

/*****
Function name: CRegistry::QueryInfo
Description   :
Return type   : BOOL
Argument      : void
*****/
BOOL CRegistry::QueryInfo( void )
{
    _ASSERTE( this );
    TCHAR class_name[ 2048 ];
    ::ZeroMemory( class_name, sizeof( class_name ) );
    DWORD size_of_class_name = (sizeof(class_name)/sizeof(*(class_name))) - 1;

    m_ErrorCode = ::RegQueryInfoKey( m_KeyHandle,
                                    class_name,
                                    &size_of_class_name,
                                    (LPDWORD) NULL,
                                    &m_NumberOfSubkeys,
                                    &m_LongestSubkeyNameLength,
                                    &m_LongestClassNameLength,
                                    &m_NumberOfValues,
                                    &m_LongestValueNameLength,
                                    &m_LongestValueDataLength,
                                    &m_SecurityDescriptorLength,
                                    &m_LastWriteTime );

    if ( m_ErrorCode == ERROR_SUCCESS )
    {
        m_ClassName = class_name;
        return( TRUE );
    }
}

```



```

    }
    else
    {
        return( FALSE );
    }
}

/* *****
Function name: CRegistry::QueryValue
Description :
Return type : BOOL
Argument    : LPCTSTR      name_of_value
Argument1   : KeyValueTypes value_type
Argument    : LPBYTE       address_of_buffer
Argument    : DWORD&       size_of_buffer
***** */
BOOL CRegistry::QueryValue( LPCTSTR      name_of_value,
                           KeyValueTypes value_type,
                           LPBYTE       address_of_buffer,
                           DWORD&       size_of_buffer )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    /*
    ** address_of_buffer and size_of_buffer can be NULL
    */

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    // We were passed a pointer, do not trust it

    try
    {
        DWORD temp_data_type = (DWORD) value_type;

        m_ErrorCode = ::RegQueryValueEx( m_KeyHandle,
                                          (LPTSTR) name_of_value,
                                          NULL,
                                          &temp_data_type,
                                          address_of_buffer,
                                          &size_of_buffer );

        if ( m_ErrorCode == ERROR_SUCCESS )
        {
            value_type = (KeyValueTypes) temp_data_type;
            return( TRUE );
        }
        else
        {
            return( FALSE );
        }
    }
    catch( ... )
    {
        m_ErrorCode = ERROR_EXCEPTION_IN_SERVICE;
        return( FALSE );
    }
}

/* *****
Function name: CRegistry::SetBinaryValue
***** */

```

```

Description :
Return type : BOOL
Argument    : LPCTSTR name_of_value
Argument    : const BYTE bytes_to_write[]
Argument    : DWORD num_bytes_to_write

```

```

*****/
BOOL CRegistry::SetBinaryValue( LPCTSTR name_of_value, const BYTE bytes_to_write[], DWORD ✓
    num_bytes_to_write )

```

```

{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

```

```

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

```

```

    BOOL return_value = SetValue( name_of_value, typeBinary, (LPBYTE)bytes_to_write, ✓
        num_bytes_to_write );
    return( return_value );
}

```

```

/*****
Function name: CRegistry::SetDoubleWordValue
Description :
Return type : BOOL
Argument    : LPCTSTR name_of_value
Argument    : DWORD value_to_write
*****/

```

```

BOOL CRegistry::SetDoubleWordValue( LPCTSTR name_of_value, DWORD value_to_write )

```

```

{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

```

```

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

```

```

    return( SetValue( name_of_value, typeDoubleWord, (const PBYTE) &value_to_write, sizeof(✓
        DWORD ) ) );
}

```

```

/*****
Function name: CRegistry::SetStringValue
Description :
Return type : BOOL
Argument    : LPCTSTR name_of_value
Argument    : const string& string_value
*****/

```

```

BOOL CRegistry::SetStringValue( LPCTSTR name_of_value, const string& string_value )

```

```

{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

```

```

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

```

```

    return( SetValue( name_of_value, typeString, (const PBYTE) string_value.c_str(), ✓
        string_value.size() + 1 ) );
}

```

```

/*****
Function name: CRegistry::SetValue
Description   :
Return type   : BOOL
Argument      : LPCTSTR name_of_value
Argument      : DWORD value
*****/
BOOL CRegistry::SetValue( LPCTSTR name_of_value, DWORD value )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    return( SetDoubleWordValue( name_of_value, value ) );
}

/*****
Function name: CRegistry::SetValue
Description   :
Return type   : BOOL
Argument      : LPCTSTR name_of_value
Argument      : const string& string_to_write
*****/
BOOL CRegistry::SetValue( LPCTSTR name_of_value, const string& string_to_write )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    return( SetStringValue( name_of_value, string_to_write ) );
}

/*****
Function name: CRegistry::SetValue
Description   :
Return type   : BOOL
Argument      : LPCTSTR name_of_value
Argument      : const KeyValueType type_of_value_to_set
Argument      : const PBYTE address_of_value_data
Argument      : const DWORD size_of_data
*****/
BOOL CRegistry::SetValue( LPCTSTR name_of_value,
                          const KeyValueType type_of_value_to_set,
                          const PBYTE address_of_value_data,
                          const DWORD size_of_data )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );
    _ASSERT( address_of_value_data != NULL );

    if ( name_of_value == NULL || address_of_value_data == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }
}

```

```
    // We were passed a pointer, do not trust it

    try
    {
        m_ErrorCode = ::RegSetValueEx( m_KeyHandle,
                                       name_of_value,
                                       0,
                                       type_of_value_to_set,
                                       address_of_value_data,
                                       size_of_data );

        if ( m_ErrorCode == ERROR_SUCCESS )
        {
            return( TRUE );
        }
        else
        {
            return( FALSE );
        }
    }
    catch( ... )
    {
        m_ErrorCode = ERROR_EXCEPTION_IN_SERVICE;
        return( FALSE );
    }
}
```

```

#include "stdafx.h"
#include "registryBase.h"
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CRegistryBase

CRRegistryBase::CRRegistryBase()
{
}

bool CRRegistryBase::connect(CRegistry::_Keys eKey, LPCSTR pszComputer)
{
    if (m_oRegistry.Connect(eKey) == REG_FAILURE)
    {
        string strRegName;
        m_oRegistry.GetRegistryName(strRegName);

        m_strLastError = "Unable to connect to {";
        m_strLastError += strRegName;
        m_strLastError += "}";
        if (pszComputer != NULL)
        {
            m_strLastError += " on computer {";
            m_strLastError += pszComputer;
            m_strLastError += "}";
        }
        return false;
    }

    return true;
}

bool CRRegistryBase::getValue(LPCSTR pszValueName, string & strValue)
{
    if (m_oRegistry.GetValue(pszValueName, strValue) == REG_FAILURE)
    {
        m_strLastError = "Unable to retrieve value {";
        m_strLastError += pszValueName;
        m_strLastError += "}";
        return false;
    }

    return true;
}

bool CRRegistryBase::getValue(LPCSTR pszValueName, unsigned long & lValue)
{
    if (m_oRegistry.GetValue(pszValueName, lValue) == REG_FAILURE)
    {
        m_strLastError = "Unable to retrieve value {";
        m_strLastError += pszValueName;
        m_strLastError += "}";
        return false;
    }

    return true;
}

bool CRRegistryBase::getBinaryValue(LPCSTR pszValueName, LPBYTE pBuff, DWORD * pdwBuffSize)
{
    CRegistry::_KeyValueTypes eValueTypes = CRegistry::_typeBinary;
    if (m_oRegistry.QueryValue(pszValueName, eValueTypes, pBuff,
        *pdwBuffSize) == REG_FAILURE)
    {
        m_strLastError = "Unable to retrieve value {";
        m_strLastError += pszValueName;
        m_strLastError += "}";
        return false;
    }

    return true;
}

```

```
}

int CRegistryBase::enumKeys(vector<string> & aryKeys)
{
    DWORD dwIdx = 0;
    string strKeyName;
    string strClassName;
    while (m_oRegistry.EnumerateKeys(dwIdx++, strKeyName, strClassName))
        aryKeys.push_back(strKeyName);
    return aryKeys.size();
}

bool CRegistryBase::setValue(LPCSTR pszValueName, LPCSTR pszValue)
{
    if (m_oRegistry.SetValue(pszValueName, CRegistry::typeString, (PBYTE) pszValue,
        strlen(pszValue) + 1) == REG_FAILURE)
    {
        m_strLastError = "Unable to write value [";
        m_strLastError += pszValueName;
        m_strLastError += "]\n";
        return false;
    }

    return true;
}

bool CRegistryBase::setValue(LPCSTR pszValueName, unsigned long lValue)
{
    if (m_oRegistry.SetValue(pszValueName, CRegistry::typeDoubleWord, (PBYTE) &lValue,
        sizeof(lValue)) == REG_FAILURE)
    {
        m_strLastError = "Unable to write value [";
        m_strLastError += pszValueName;
        m_strLastError += "]\n";
        return false;
    }

    return true;
}

bool CRegistryBase::openKey(LPCSTR pszKeyPath)
{
    if (m_oRegistry.Open(pszKeyPath, CRegistry::permissionAllAccess) == REG_FAILURE)
    {
        m_strLastError = "Unable to open key [";
        m_strLastError += pszKeyPath;
        m_strLastError += "]\n";
        return false;
    }

    return true;
}

bool CRegistryBase::createKey(LPCSTR pszKeyPath)
{
    CRegistry::CreationDisposition eDisposition;

    if (m_oRegistry.Create(pszKeyPath,
        NULL,
        CRegistry::optionsNonVolatile,
        CRegistry::permissionAllAccess,
        NULL,
        &eDisposition) == REG_FAILURE)
    {
        m_strLastError = "Unable to create key [";
        m_strLastError += pszKeyPath;
        m_strLastError += "]\n";
        return false;
    }
}
```

```
    )  
    return true;  
}
```

```
#ifndef _registryBase_h
#define _registryBase_h

#include "registry.h"

class CRegistryBase
{
protected:
    CRegistry          m_oRegistry;

    enum {REG_FAILURE = 0, REG_SUCCESS = 1};

    CRegistryBase();
    bool connect(CRegistry::_Keys eKey, LPCSTR pszComputer = NULL);
    bool getValue(LPCSTR pszValueName, string & strValue);
    bool getValue(LPCSTR pszValueName, unsigned long & lValue);
    bool getBinaryValue(LPCSTR pszValueName, LPBYTE pBuff, DWORD * pdwBuffSize);
    bool setValue(LPCSTR pszValueName, LPCSTR pszValue);
    bool setValue(LPCSTR pszValueName, unsigned long lValue);
    bool openKey(LPCSTR pszKeyPath);
    bool createKey(LPCSTR pszKeyPath);
    int enumKeys(vector<string> & aryKeys);
    bool close() { return (m_oRegistry.Close() == TRUE); }

public:
    string          m_strLastError;
};

#endif
```



```
//{{NO_DEPENDENCIES}}
// Microsoft Visual C++ generated include file.
// Used by LCKioskClient.rc
//
#define IDS_SERVICENAME 100

#define IDR_LCKioskClient 100

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifdef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE 201
#define _APS_NEXT_COMMAND_VALUE 32768
#define _APS_NEXT_CONTROL_VALUE 201
#define _APS_NEXT_SYMED_VALUE 101
#endif
#endif
```

```
// stdafx.cpp : source file that includes just the standard includes
//  stdafx.pch will be the pre-compiled header
//  stdafx.obj will contain the pre-compiled type information
```

```
#include "stdafx.h"
```

```
#ifdef _ATL_STATIC_REGISTRY
```

```
#include <statreg.h>
```

```
#include <statreg.cpp>
```

```
#endif
```

```
#include <atlimpl.cpp>
```

```
#include "stdafx.h"
#include "threadMain.h"
#include "threadMonitor.h"

#include "filenameDelimited.h"
#include "registryKClient.h"

BOOL CKioskClientApp::InitInstance()
{
    m_pthreadMonitor = new CThreadMonitor(this);
    BOOL fSuccess = m_pthreadMonitor->CreateThread(0);
    if (!fSuccess)
    {
        CLogMsgEvent msg(LCEV_GENERIC, SVRTY_ERROR);
        msg << "Unable to create monitor thread in CKioskClientApp::InitInstance()";
        msg.Post(_logAll);
    }

    return fSuccess;
}

int CKioskClientApp::ExitInstance()
{
    m_pthreadMonitor->PostThreadMessage(WM_QUIT, 0L, 0L);
    return 0;
}

int CKioskClientApp::Run()
{
    return CWinThread::Run();
}
```

```
// threadMonitor.cpp : implementation file
//
```

```
#include "stdafx.h"
#include "lckioskclient.h"
#include "threadMonitor.h"
#include "wndMonitorISP.h"
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

```
////////////////////////////////////
// CThreadMonitor
```

```
string CThreadMonitor::m_strWndClass;
```

```
IMPLEMENT_DYNCREATE(CThreadMonitor, CWinThread)
```

```
CThreadMonitor::CThreadMonitor(CWinThread * pthreadParent)
{
    m_pthreadParent = pthreadParent;
    m_fError = false;
}
```

```
CThreadMonitor::~CThreadMonitor()
{
}
```

```
BOOL CThreadMonitor::InitInstance()
```

```
{
    if (m_strWndClass.size() == 0)
        m_strWndClass = AfxRegisterWndClass(0);

    // todo: instantiate a window here of type CWndMonitor. now the only type we have is
    // CWndMonitorISP. Derive new CWndMonitors has needed. ex: CWndMonitorDSL

    m_pMainWnd = new CWndMonitorISP();
    ((CWndMonitorISP *)m_pMainWnd)->m_pthreadMonitor = this;
    BOOL fSuccess = m_pMainWnd->CreateEx(0, m_strWndClass.c_str(), "wndInternet", WS_POPUP,
    '    CRect(0,0,0,0), NULL, 0);

    if (!fSuccess)
    {
        CLogMsgEvent msg(LCEV_GENERIC, SVRTY_ERROR);
        msg << "Unable to create Monitor's main window in CThreadMonitor::InitInstance()";
        msg.Post(_logAll);
        m_fError = true;
        m_pthreadParent->PostThreadMessage(WM_QUIT, 0, 0);
    }

    return fSuccess;
}
```

```
int CThreadMonitor::ExitInstance()
{
    delete m_pMainWnd;
    return CWinThread::ExitInstance();
}
```

```
BEGIN_MESSAGE_MAP(CThreadMonitor, CWinThread)
//{{AFX_MSG_MAP(CThreadMonitor)
    // NOTE - the ClassWizard will add and remove mapping macros here.
//}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()
```

```
// CThreadMonitor message handlers
```

```

#if !defined(AFX_WNDMONITOR_H__32F92C5B_E2F7_11D3_B884_76D29F000000__INCLUDED_)
#define AFX_WNDMONITOR_H__32F92C5B_E2F7_11D3_B884_76D29F000000__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// wndMonitor.h : header file
//

class CThreadMonitor;
////////////////////////////////////
// CWndMonitor window

#include "registryKClient.h"

class CWndMonitor : public CWnd
{
    friend class CThreadMonitor;
// Construction
public:
    CWndMonitor();

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CWndMonitor)
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CWndMonitor();

    // Generated message map functions
protected:
    //{{AFX_MSG(CWndMonitor)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()

protected:
    unsigned long          m_lProcInterval;
    CThreadMonitor *       m_pthreadMonitor;
    CRegistryKClient       m_registryKClient;
};

////////////////////////////////////
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif // !defined(AFX_WNDMONITOR_H__32F92C5B_E2F7_11D3_B884_76D29F000000__INCLUDED_)

```

```
#if !defined(AFX_WNDMONITORISP_H__32F92C5C_E2F7_11D3_B884_76D29F000000__INCLUDED_)
#define AFX_WNDMONITORISP_H__32F92C5C_E2F7_11D3_B884_76D29F000000__INCLUDED_
```

```
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// wndMonitorISP.h : header file
//
```

```
#include "wndMonitor.h"
#include "dialer.h"
```

```
////////////////////////////////////
// CWndMonitorISP window
```

```
class CWndMonitorISP : public CWndMonitor
```

```
{
// Construction
public:
```

```
    CWndMonitorISP();
```

```
// Attributes
public:
```

```
// Operations
public:
```

```
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CWndMonitorISP)
//}}AFX_VIRTUAL
```

```
// Implementation
public:
    virtual ~CWndMonitorISP();
```

```
    // Generated message map functions
```

```
protected:
    //{{AFX_MSG(CWndMonitorISP)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnTimer(UINT nIDEvent);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
```

```
protected:
    static UINT m_nRasDialMsg;
```

```
    enum { RETRY_FOREVER = -1 };
```

```
    enum { TIMER_CHECKTIME = 1, // check time for processing
          TIMER_RETRY }; // retry a failed connect
```

```
    enum { INTERVAL_CHECKTIME = 60000 }; // how often to look at the time for processing
```

```
    // for debugging purposes
```

```
    enum State {ST_NONE, ST_CHECKTIME, ST_CONNECTING, ST_RETRYING, ST_EXCHANGING, ST_APPLYING};
```

```
    State m_ePreviousState;
    State m_eState;
```

```
    int m_nRetryCount;
    bool m_fUseRas;
    bool m_fExchangeDone;
    CDialer * m_pdialer;
    CDialerRAS * m_pdialerRas;
    CDialerWinInet * m_pdialerWinInet;
```

```
LRESULT onTimerCheckTime(WPARAM wParam, LPARAM lParam);
bool onTimerRetry();
LRESULT onExchange(WPARAM wParam, LPARAM lParam);
LRESULT onApply(WPARAM wParam, LPARAM lParam);
LRESULT onTryConnect(WPARAM wParam, LPARAM lParam);
bool exchangeFiles();
bool deleteOldBackups();
void setState(State eNewState){m_ePreviousState = m_eState;m_eState = eNewState;}
static bool _funcSortFileName(string & str1, string & str2);

afx_msg LRESULT onRasDialReport(WPARAM wRasConnState, LPARAM dwError);
afx_msg LRESULT onDialConnect(WPARAM wParam, LPARAM dwError);
};
```

```
////////////////////////////////////
```

```
//({AFX_INSERT_LOCATION})
```

```
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.
```

```
#endif // !defined(AFX_WNDMONITORISF_H__32F92C5C_E2F7_11D3_B884_76D29F000000_INCLUDED_)
```



```
#ifndef _zipUtil_h
#define _zipUtil_h

class CZipUtil
{
public:
    enum { ZF_OverWrite = 0x0001,
           ZF_UseDirectoryNames = 0x0002 };
public:
    CZipUtil();
    virtual ~CZipUtil();
    virtual bool unzipFile(LPCSTR pszZipFile, LPCSTR pszTargetDir, unsigned short nFlags) ✓
        = 0;
};

class CZipUtilXceed : public CZipUtil
{
protected:
    XZip::IXceedZipPtr      m_spZip;
public:
    CZipUtilXceed();
    virtual bool unzipFile(LPCSTR pszZipFile, LPCSTR pszTargetDir, unsigned short nFlags);
};

#endif
```

```
#include "stdafx.h"
#include "zipUtil.h"

CZipUtil::CZipUtil()
{
}

CZipUtil::~CZipUtil()
{
}

CZipUtilXceed::CZipUtilXceed()
{
    HRESULT hr = m_spZip.CreateInstance(__uuidof(XZip::XceedZip));
    if (FAILED(hr))
    {
        CLogMsgEvent msg(LCEV_GENERIC, SVRTY_WARNING);
        msg << "Unable to Instantiate XceedZip. Error = [0x" << std::hex << hr << "].";
        msg.Post(_logAll);
    }
}

bool CZipUtilXceed::unzipFile(LPCSTR pszZipFile, LPCSTR pszTargetDir, unsigned short nFlags)
{
    bool fSuccess = true;

    m_spZip->ZipFilename = pszZipFile;
    m_spZip->UnzipToFolder = pszTargetDir;
    m_spZip->PreservePaths = (nFlags & ZF_UseDirectoryNames) ? TRUE : FALSE;
    m_spZip->SkipIfExists = (nFlags & ZF_OverWrite) ? FALSE : TRUE;
    XZip::xcdError eErrorCode = m_spZip->Unzip();
    if (eErrorCode != XZip::xerSuccess)
    {
        CLogMsgEvent msg(LCEV_GENERIC, SVRTY_WARNING);
        msg << "Error occurred while unzipping file [" << pszZipFile << "] to directory [";
        msg << pszTargetDir << "]. Xceed Error Code = [" << (long) eErrorCode << "].";
        msg.Post(_logAll);
        fSuccess = false;
    }

    return fSuccess;
}
```

```
// Encryptor.cpp: implementation of the CEncryptor class.
```

```
//
```

```
////////////////////////////////////
```

```
#include "StdAfx.h"
```

```
#include "Encryptor.h"
```

```
#include <time.h>
```

```
#ifdef _DEBUG
```

```
#undef THIS_FILE
```

```
static char THIS_FILE[] = __FILE__;
```

```
#endif
```

```
////////////////////////////////////
```

```
// Construction/Destruction
```

```
////////////////////////////////////
```

```
CEncryptor::CEncryptor()
```

```
{
```

```
    m_strDefaultKey = "phepmagi";
```

```
}
```

```
/* **** */
```

```
    FUNCTION: Encrypt
```

```
    CLASS: CEncryptor
```

```
DESCRIPTION: Encrypts an ascii string into a series of ascii hex digits.
```

```
PARAMETERS: pszIn - pointer to string to encrypt
```

```
    pszKey - encryption key, this parameter may be null, in which  
             case a default encryption key is used.
```

```
    strOut - reference to a string that will receive the encryption  
             results.
```

```
RETURNS: true on success
```

```
        false on error
```

```
/* **** */
```

```
bool CEncryptor::Encrypt(LPCTSTR pszIn, LPCSTR pszKey, string & strOut)
```

```
{
```

```
    strOut = "";
```

```
    string strKey = pszKey == NULL ? m_strDefaultKey : pszKey;
```

```
    int nKeyLen = strKey.size();
```

```
    int nKeyPos = -1;
```

```
    srand((unsigned)time(NULL));
```

```
    int nOffset = rand() % 255;
```

```
    char pszBuff [4];
```

```
    sprintf(pszBuff, "%02x", nOffset);
```

```
    strOut += pszBuff;
```

```
    char chKey;
```

```
    for (int i = 0; pszIn[i] != 0; i++)
```

```
    {
```

```
        int nSrcAscii = (pszIn[i] + nOffset) % 255;
```

```
        if (nKeyPos < nKeyLen - 1)
```

```
        {
```

```
            nKeyPos++;
```

```
            chKey = strKey[nKeyPos];
```

```
        }
```

```
        else
```

```
        {
```

```
            nKeyPos = -1;
```

```
        }
```

```

        chKey = 0;
    }

    nSrcAscii ^= chKey;
    sprintf(pszBuff, "%02x", nSrcAscii);
    strOut += pszBuff;
    nOffset = nSrcAscii;
}

return true;
}

/*****
FUNCTION: Decrypt

CLASS: CEncryptor

DESCRIPTION: Given encrypted data, decrypts back to its original form.

PARAMETERS: pszIn    - pointer to encrypted data.
             pszKey   - pointer to key that was used to encrypt the data. This
                       may be NULL in which case a default key is used.
             strOut   - reference to a string that will receive the decrypted
                       data.

RETURNS: true - no errors
        false - an error occurred
*****/
bool CEncryptor::Decrypt(LPCTSTR pszIn, LPCSTR pszKey, string & strOut)
{
    string strKey = pszKey == NULL ? "" : pszKey;
    if (strKey.size() == 0)
        strKey = m_strDefaultKey;

    strOut = "";

    int nSrcPos = 2;
    int nKeyPos = -1;
    string strSrc = pszIn;
    int nSrcLen = strSrc.size();
    int nKeyLen = strKey.size();
    int nSrcAscii = 0;
    int nTmpSrcAscii = 0;

    int nOffset;
    if (AsciiHexToInt(strSrc.substr(0, 2), &nOffset))
        return false;

    char chKey;

    do
    {
        if (AsciiHexToInt(strSrc.substr(nSrcPos, 2), &nSrcAscii))
            return false;

        if (nKeyPos < nKeyLen - 1)
        {
            nKeyPos += 1;
            chKey = strKey[nKeyPos];
        }
        else
        {
            nKeyPos = -1;
            chKey = 0;
        }
    }

```

```

        nTmpSrcAscii = nSrcAscii ^ chKey;

        if (nTmpSrcAscii <= nOffset)
            nTmpSrcAscii += 255 - nOffset;
        else
            nTmpSrcAscii -= nOffset;

        strOut += (char)nTmpSrcAscii;
        nOffset = nSrcAscii;
        nSrcPos += 2;
    } while (nSrcPos < nSrcLen);

    return true;
}

/*****
FUNCTION: AsciiHexToInt

CLASS: CEncryptor

DESCRIPTION: Helper function that takes a string of ascii hex digits
             (ie. "EF34DC") and returns the binary decimal representation.

PARAMETERS: pszString - ascii hex digits to convert
            pnAnswer - pointer to an int that will receive the conversion.

RETURNS: true on error
         false on success
*****/
short CEncryptor::AsciiHexToInt(LPCTSTR pszString, int * pnAnswer)
{
    int nPlaces = strlen(pszString) - 1;

    short    wError = FALSE;
    char      cWork;
    int nAnswer = 0;

    for (int i = 0; !wError && (cWork = pszString[i]) != 0; i++)
    {
        cWork = toupper(cWork);
        if (!isdigit(cWork))
        {
            cWork -= 'A' - 10;
            if (cWork < 0 || cWork > 15)
                wError = TRUE;
        }
        else
            cWork &= 0x0f;

        if (nPlaces)
            nAnswer += cWork * (nPlaces-- * 16);
        else
            nAnswer += cWork;
    }

    *pnAnswer = nAnswer;
    return wError;
}

```

```
/*//////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////
Encrypt/ Decrypt Routines
//////////////////////////////////////////////////////////////////
```

Dependencies :

```
#include <string>
#include <list>
#include <fstream>
#include <strstream>
using namespace std;
```

```
//////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////
```

```
#ifndef _ENCRYPTOR_H
#define _ENCRYPTOR_H

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
```

```
class CEncryptor
{
protected:
    string m_strDefaultKey;

    short AsciiHexToInt( LPCTSTR pszString, int* pnAnswer );
    short AsciiHexToInt( string& strIn, int* pnAnswer )
        {return AsciiHexToInt(strIn.c_str(), pnAnswer);}

public:
    CEncryptor();
    bool Encrypt(LPCTSTR pszIn, LPCSTR psKey, string & strOut);
    bool Decrypt(LPCTSTR pszIn, LPCSTR psKey, string & strOut);
};
```

```
#endif // _ENCRYPTOR_H
```

```
#include "stdafx.h"
#include "filenameDelimited.h"

CFileNameDelimited::CFileNameDelimited()
{
    m_bDelimiter = '_';
}

bool CFileNameDelimited::append(LPCSTR pszFieldName, LPCSTR pszFieldValue)
{
    FILENAME_FIELD rField;

    rField.strName = pszFieldName;
    rField.strValue = pszFieldValue;

    push_back(rField);

    return true;
}

bool CFileNameDelimited::append(LPCSTR pszFieldName, long lFieldValue)
{
    char pszValue [20];
    ltoa(lFieldValue, pszValue, 10);
    return append(pszFieldName, pszValue);
}

bool CFileNameDelimited::append(LPCSTR pszFieldName, DATE dateValue, LPCSTR pszDateFormat)
{
    COleDateTime odt(dateValue);

    char * pszFormat = (char *) pszDateFormat;
    if (pszFormat == NULL)
        pszFormat = "%m%d%y";

    append(pszFieldName, (LPCSTR) odt.Format(pszFormat));

    return true;
}

int CFileNameDelimited::getIndex(LPCSTR pszFieldName)
{
    int nCount = size();

    for (int i = 0; i < nCount; i++)
    {
        if ((*this)[i].strName.compare(pszFieldName) == 0)
            return i;
    }

    return -1;
}

bool CFileNameDelimited::get(int nIdx, string & strValue)
{
    strValue = "";

    if (nIdx < 0 || nIdx >= size())
        return false;

    strValue = (*this)[nIdx].strValue;

    return true;
}

bool CFileNameDelimited::get(int nIdx, long & lFieldValue)
```

```
{
    bool fSuccess;

    string strValue;

    if (fSuccess = get(nIdx, strValue))
        lFieldValue = atol(strValue.c_str());
    else
        lFieldValue = 0;

    return fSuccess;
}

bool CFileNameDelimited::get(int nIdx, DATE & dateValue)
{
    bool fSuccess;

    string strValue;

    if (fSuccess = get(nIdx, strValue))
    {
        strValue.insert(4, "/");
        strValue.insert(2, "/");
        COleDateTime odt;
        odt.ParseDateTime(strValue.c_str());
        dateValue = (DATE) odt;
    }
    else
        dateValue = 0.0;

    return fSuccess;
}

bool CFileNameDelimited::set(int nIdx, LPCSTR pszValue)
{
    if (nIdx < 0)
        return false;

    // pad out vector up to occurrence referenced
    if (nIdx >= size())
    {
        for (int i = size(); i <= nIdx; i++)
            append("", "");
    }

    (*this)[nIdx].strValue = pszValue;

    return true;
}

bool CFileNameDelimited::set(int nIdx, long lFieldValue)
{
    char pszValue [20];
    ltoa(lFieldValue, pszValue, 10);
    return set(nIdx, pszValue);
}

bool CFileNameDelimited::set(int nIdx, DATE dateValue, LPCSTR pszDateFormat)
{
    COleDateTime odt(dateValue);

    char * pszFormat = (char *) pszDateFormat;
    if (pszFormat == NULL)
        pszFormat = "%m%d%y";

    return set(nIdx, (LPCSTR) odt.Format(pszDateFormat));
}
```



```
bool CFileNameDelimited::setFullName(LPCSTR pszFileName, bool fClear)
{
    string strValue;

    if (fClear)
        clear();

    string strName = pszFileName;

    // pick out the extension if it exist
    int nExtPos = strName.find_last_of('.');
    if (nExtPos != string::npos)
    {
        m_strExtension = strName.substr(nExtPos + 1, strName.size() - nExtPos);
        strName.resize(nExtPos);
    }

    int nIdx = 0;

    // parse the name out into fields and set them
    if (strName.size())
    {
        int nLastPos = 0;
        int nNewPos = 0;

        while ((nNewPos = strName.find(m_bDelimiter, nLastPos)) != string::npos)
        {
            strValue = strName.substr(nLastPos, nNewPos - nLastPos);
            set(nIdx++, strValue.c_str());
            nLastPos = nNewPos + 1;
        }

        strValue = strName.substr(nLastPos, nNewPos - nLastPos);
        set(nIdx++, strValue.c_str());
    }

    // if file name shorter than fields, clear values on fields
    int nSize = size();
    for (; nIdx < nSize; nIdx++)
        set(nIdx, "");

    return true;
}

void CFileNameDelimited::setExtension(LPCSTR pszExt)
{
    m_strExtension = pszExt;
}

bool CFileNameDelimited::getFullName(string & strFileName)
{
    strFileName = "";

    CFileNameDelimited::iterator it;
    for (it = begin(); it != end(); it++)
    {
        if (strFileName.size())
            strFileName += m_bDelimiter;

        strFileName += (*it).strValue;
    }

    if (m_strExtension.size())
    {
        strFileName += ".";
        strFileName += m_strExtension;
    }
}
```

```
    }

    return true;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// CFileNameKiosk

CFileNameKiosk::CFileNameKiosk()
{
    append("direction", "");
    append("kiosk_id", "");
    append("date", "");
}
```

```
#ifndef _filenameDelimited_h
#define _filenameDelimited_h

struct FILENAME_FIELD
{
    string      strName;
    string      strValue;

    FILENAME_FIELD & operator=(const FILENAME_FIELD & rField)
    {
        strName = rField.strName;
        strValue = rField.strValue;
        return *this;
    }
};

class CFileNameDelimited : public vector<FILENAME_FIELD>
{
protected:
    char          m_bDelimiter;
    string         m_strExtension;

public:
    CFileNameDelimited();
    bool append(LPCSTR pszFieldName, LPCSTR pszFieldValue);
    bool append(LPCSTR pszFieldName, long lFieldValue);
    bool append(LPCSTR pszFieldName, DATE dateValue, LPCSTR pszDateFormat = NULL);

    bool get(LPCSTR pszFieldName, string & strValue);
    bool get(LPCSTR pszFieldName, long & lFieldValue);
    bool get(LPCSTR pszFieldName, DATE & dateValue);

    bool set(LPCSTR pszFieldName, LPCSTR pszValue);
    bool set(LPCSTR pszFieldName, long lFieldValue);
    bool set(LPCSTR pszFieldName, DATE dateValue, LPCSTR pszDateFormat = NULL);

    bool get(int nIndex, string & strValue);
    bool get(int nIndex, long & lFieldValue);
    bool get(int nIndex, DATE & dateValue);

    bool set(int nIndex, LPCSTR pszValue);
    bool set(int nIndex, long lFieldValue);
    bool set(int nIndex, DATE dateValue, LPCSTR pszDateFormat = NULL);

    bool setFullName(LPCSTR pszFileName, bool fClear = false);
    void setExtension(LPCSTR pszExt);

    bool getFullName(string & strFileName);

    int getIndex(LPCSTR pszFieldName);
};

inline bool CFileNameDelimited::get(LPCSTR pszFieldName, string & strValue)
{
    return get(getIndex(pszFieldName), strValue);
}

inline bool CFileNameDelimited::get(LPCSTR pszFieldName, long & lFieldValue)
{
    return get(getIndex(pszFieldName), lFieldValue);
}

inline bool CFileNameDelimited::get(LPCSTR pszFieldName, DATE & dateValue)
{
    return get(getIndex(pszFieldName), dateValue);
}
```

```
inline bool CFileNameDelimited::set(LPCSTR pszFieldName, LPCSTR pszValue)
{
    return set(getIndex(pszFieldName), pszValue);
}
```

```
inline bool CFileNameDelimited::set(LPCSTR pszFieldName, long lFieldValue)
{
    return set(getIndex(pszFieldName), lFieldValue);
}
```

```
inline bool CFileNameDelimited::set(LPCSTR pszFieldName, DATE dateValue, LPCSTR
    pszDateFormat)
{
    return set(getIndex(pszFieldName), dateValue, pszDateFormat);
}
```

```
////////////////////////////////////
```

```
// CFileNameKiosk
```

```
class CFileNameKiosk : public CFileNameDelimited
```

```
{
```

```
public:
```

```
    CFileNameKiosk();
```

```
};
```

```
#endif
```

```
// LCKioskServer.cpp : Implementation of WinMain
```

```
// Note: Proxy/Stub Information
//      To build a separate proxy/stub DLL,
//      run nmake -f LCKioskServerps.mk in the project directory.
```

```
#include "stdafx.h"
#include "resource.h"
#include <initguid.h>
#include "threadMain.h"
#include "LCKioskServer.h"
```

```
#include "LCKioskServer_i.c"
```

```
#include <stdio.h>
```

```
////////////////////////////////////
// MFC support
```

```
CKioskServerApp _theApp;
```

```
////////////////////////////////////
// ATL support
```

```
CServiceModule _Module;
```

```
////////////////////////////////////
//Global declarations
```

```
CLogNTEvents    _logEvents("Kiosk Server");
CLogFile         _logFile("c:\\LCKioskServer.log");
CLogDebug        _logDebug;
CLogMulti        _logAll;
```

```
BEGIN_OBJECT_MAP(ObjectMap)
END_OBJECT_MAP()
```

```
LPCTSTR FindOneOf(LPCTSTR p1, LPCTSTR p2)
```

```
{
    while (p1 != NULL && *p1 != NULL)
    {
        LPCTSTR p = p2;
        while (p != NULL && *p != NULL)
        {
            if (*p1 == *p)
                return CharNext(p1);
            p = CharNext(p);
        }
        p1 = CharNext(p1);
    }
    return NULL;
}
```

```
// Although some of these functions are big they are declared inline since they are only used once ✓
```

```
inline HRESULT CServiceModule::RegisterServer(BOOL bRegTypeLib, BOOL bService)
```

```
{
    HRESULT hr = CoInitialize(NULL);
    if (FAILED(hr))
        return hr;

    // Remove any previous service since it may point to
    // the incorrect file
    Uninstall();

    // Add service entries
```

```
UpdateRegistryFromResource(IDR_LCKioskServer, TRUE);

// Adjust the AppID for Local Server or Service
CRegKey keyAppID;
LONG lRes = keyAppID.Open(HKEY_CLASSES_ROOT, _T("AppID"), KEY_WRITE);
if (lRes != ERROR_SUCCESS)
    return lRes;

CRegKey key;
lRes = key.Open(keyAppID, _T("{BF823564-E93B-11D3-B88C-CC792E000000}"), KEY_WRITE);
if (lRes != ERROR_SUCCESS)
    return lRes;
key.DeleteValue(_T("LocalService"));

if (bService)
{
    key.SetValue(_T("LCKioskServer"), _T("LocalService"));
    key.SetValue(_T("-Service"), _T("ServiceParameters"));
    // Create service
    Install();
}

// Add object entries
hr = CComModule::RegisterServer(bRegTypeLib);

CoUninitialize();
return hr;
}

inline HRESULT CServiceModule::UnregisterServer()
{
    HRESULT hr = CoInitialize(NULL);
    if (FAILED(hr))
        return hr;

    // Remove service entries
    UpdateRegistryFromResource(IDR_LCKioskServer, FALSE);
    // Remove service
    Uninstall();
    // Remove object entries
    CComModule::UnregisterServer(TRUE);
    CoUninitialize();
    return S_OK;
}

inline void CServiceModule::Init(_ATL_OBJMAP_ENTRY* p, HINSTANCE h, UINT nServiceNameID,
    const GUID* plibid)
{
    CComModule::Init(p, h, plibid);

    m_bService = TRUE;

    LoadString(h, nServiceNameID, m_szServiceName, sizeof(m_szServiceName) / sizeof
        (TCHAR));

    // set up the initial service status
    m_hServiceStatus = NULL;
    m_status.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
    m_status.dwCurrentState = SERVICE_STOPPED;
    m_status.dwControlsAccepted = SERVICE_ACCEPT_STOP;
    m_status.dwWin32ExitCode = 0;
    m_status.dwServiceSpecificExitCode = 0;
    m_status.dwCheckpoint = 0;
    m_status.dwWaitHint = 0;
}

LONG CServiceModule::Unlock()
```

```
{
    LONG l = CComModule::Unlock();
    if (l == 0 && !m_bService)
        PostThreadMessage(dwThreadId, WM_QUIT, 0, 0);
    return l;
}

BOOL CServiceModule::IsInstalled()
{
    BOOL bResult = FALSE;

    SC_HANDLE hSCM = ::OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);

    if (hSCM != NULL)
    {
        SC_HANDLE hService = ::OpenService(hSCM, m_szServiceName, SERVICE_QUERY_CONFIG);
        if (hService != NULL)
        {
            bResult = TRUE;
            ::CloseServiceHandle(hService);
        }
        ::CloseServiceHandle(hSCM);
    }
    return bResult;
}

inline BOOL CServiceModule::Install()
{
    if (IsInstalled())
        return TRUE;

    SC_HANDLE hSCM = ::OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
    if (hSCM == NULL)
    {
        MessageBox(NULL, _T("Couldn't open service manager"), m_szServiceName, MB_OK);
        return FALSE;
    }

    // Get the executable file path
    TCHAR szFilePath[_MAX_PATH];
    ::GetModuleFileName(NULL, szFilePath, _MAX_PATH);

    SC_HANDLE hService = ::CreateService(
        hSCM, m_szServiceName, m_szServiceName,
        SERVICE_ALL_ACCESS, SERVICE_WIN32_OWN_PROCESS,
        SERVICE_DEMAND_START, SERVICE_ERROR_NORMAL,
        szFilePath, NULL, NULL, _T("RPCSS\0"), NULL, NULL);

    if (hService == NULL)
    {
        ::CloseServiceHandle(hSCM);
        MessageBox(NULL, _T("Couldn't create service"), m_szServiceName, MB_OK);
        return FALSE;
    }

    ::CloseServiceHandle(hService);
    ::CloseServiceHandle(hSCM);
    return TRUE;
}

inline BOOL CServiceModule::Uninstall()
{
    if (!IsInstalled())
        return TRUE;

    SC_HANDLE hSCM = ::OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
```

```

if (hSCM == NULL)
{
    MessageBox(NULL, _T("Couldn't open service manager"), m_szServiceName, MB_OK);
    return FALSE;
}

SC_HANDLE hService = ::OpenService(hSCM, m_szServiceName, SERVICE_STOP | DELETE);

if (hService == NULL)
{
    ::CloseServiceHandle(hSCM);
    MessageBox(NULL, _T("Couldn't open service"), m_szServiceName, MB_OK);
    return FALSE;
}

SERVICE_STATUS status;
::ControlService(hService, SERVICE_CONTROL_STOP, &status);

BOOL bDelete = ::DeleteService(hService);
::CloseServiceHandle(hService);
::CloseServiceHandle(hSCM);

if (bDelete)
    return TRUE;

MessageBox(NULL, _T("Service could not be deleted"), m_szServiceName, MB_OK);
return FALSE;
}

```

```
// Logging functions
void CServiceModule::LogEvent(LPCTSTR pFormat, ...)
```

```
TCHAR chMsg[2048];
va_list pArg;

va_start(pArg, pFormat);
_vstprintf(chMsg, pFormat, pArg);
va_end(pArg);

CLogMsgEvent(LCEV_GENERIC, -1, chMsg).Post(_logAll);
}
```

//////////////////////////////////////
////

```

// Service startup and registration
inline void CServiceModule::Start()
{
    SERVICE_TABLE_ENTRY st[] =
    {
        { m_szServiceName, _ServiceMain },
        { NULL, NULL }
    };
    if (m_bService && !::StartServiceCtrlDispatcher(st))
    {
        m_bService = FALSE;
    }
    if (m_bService == FALSE)
        Run();
}

```

```
inline void CServiceModule::ServiceMain(DWORD /* dwArgc */, LPTSTR* /* lpzArgv */)
{
    // Register the control request handler
    m_status.dwCurrentState = SERVICE_START_PENDING;
    m_hServiceStatus = RegisterServiceCtrlHandler(m_szServiceName, _Handler);
    if (m_hServiceStatus == NULL)
    {

```



```
        CLogMsgEvent("Handler not installed").Post(_logAll);
        return;
    }
    SetServiceStatus(SERVICE_START_PENDING);

    m_status.dwWin32ExitCode = S_OK;
    m_status.dwCheckPoint = 0;
    m_status.dwWaitHint = 0;

    // When the Run function returns, the service has stopped.
    Run();

    SetServiceStatus(SERVICE_STOPPED);
}

inline void CServiceModule::Handler(DWORD dwOpcode)
{
    switch (dwOpcode)
    {
    case SERVICE_CONTROL_STOP:
        SetServiceStatus(SERVICE_STOP_PENDING);
        PostThreadMessage(dwThreadId, WM_QUIT, 0, 0);
        break;
    case SERVICE_CONTROL_PAUSE:
        break;
    case SERVICE_CONTROL_CONTINUE:
        break;
    case SERVICE_CONTROL_INTERROGATE:
        break;
    case SERVICE_CONTROL_SHUTDOWN:
        break;
    default:
        CLogMsgEvent("Bad service request").Post(_logAll);
    }
}

void WINAPI CServiceModule::_ServiceMain(DWORD dwArgc, LPTSTR* lpszArgv)
{
    _Module.ServiceMain(dwArgc, lpszArgv);
}

void WINAPI CServiceModule::_Handler(DWORD dwOpcode)
{
    _Module.Handler(dwOpcode);
}

void CServiceModule::SetServiceStatus(DWORD dwState)
{
    m_status.dwCurrentState = dwState;
    ::SetServiceStatus(m_hServiceStatus, &m_status);
}

void CServiceModule::Run()
{
    _Module.dwThreadId = GetCurrentThreadId();

    HRESULT hr = CoInitializeEx(NULL, COINIT_MULTITHREADED);
    if (FAILED(hr))
    {
        CLogMsgEvent msg(LCEV_GENERIC, SVRTY_ERROR);
        msg << "CoInitializeEx() failed. Error = {0x" << std::hex << hr << "}";
        msg.Post(_logAll);
        return;
    }

    // This provides a NULL DACL which will allow access to everyone.
    CSecurityDescriptor sd;
    sd.InitializeFromThreadToken();
}
```

```
hr = CoInitializeSecurity(sd, -1, NULL, NULL,
    RPC_C_AUTHN_LEVEL_PKT, RPC_C_IMP_LEVEL_IMPERSONATE, NULL, EOAC_NONE, NULL);
_ASSERTE(SUCCEEDED(hr));

hr = _Module.RegisterClassObjects(CLSCTX_LOCAL_SERVER | CLSCTX_REMOTE_SERVER,
    REGCLS_MULTIPLEUSE);
_ASSERTE(SUCCEEDED(hr));

////////////////////////////////////////
// MFC support
if (_theApp.InitApplication() == FALSE)
{
    CLogMsgEvent msg(LCEV_GENERIC, SVRTY_ERROR);
    msg << "_theApp.InitApplication() failed";
    msg.Post(_logAll);
    return;
}

if (_theApp.InitInstance() == FALSE)
{
    CLogMsgEvent msg(LCEV_GENERIC, SVRTY_ERROR);
    msg << "_theApp.InitInstance() failed";
    msg.Post(_logAll);
    _theApp.ExitInstance();
    return;
}

// end MFC support
////////////////////////////////////////

CLogMsgEvent("Service started").Post(_logAll);
if (m_bService)
    SetServiceStatus(SERVICE_RUNNING);

_theApp.Run();

_theApp.ExitInstance();

CLogMsgEvent("Service stopped").Post(_logAll);

_Module.RevokeClassObjects();

CoUninitialize();
}

////////////////////////////////////////
//
extern "C" int WINAPI _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPTSTR
    lpCmdLine,
                                int nShowCmd)
{
    _logAll.AddLog(&_logEvents);
    _logDebug.Enabled(false);
    _logAll.AddLog(&_logFile);

#ifdef _DEBUG
    _logEvents.EnableTranslation(true);
    _logDebug.Enabled(true);
    _logAll.AddLog(&_logDebug);
#endif

    lpCmdLine = GetCommandLine(); //this line necessary for _ATL_MIN_CRT
    _Module.Init(ObjectMap, hInstance, IDS_SERVICENAME, &LIBID_LCKIOSKSERVERLib);
    _Module.m_bService = TRUE;

    TCHAR szTokens[] = _T("-/");
```

```
LPCTSTR lpszToken = FindOneOf(lpCmdLine, szTokens);
while (lpszToken != NULL)
{
    if (lstrcmpi(lpszToken, _T("UnregServer"))==0)
        return _Module.UnregisterServer();

    // Register as Local Server
    if (lstrcmpi(lpszToken, _T("RegServer"))==0)
        return _Module.RegisterServer(TRUE, FALSE);

    // Register as Service
    if (lstrcmpi(lpszToken, _T("Service"))==0)
        return _Module.RegisterServer(TRUE, TRUE);

    // Initialize Configuration Registry Entries
    if (lstrcmpi(lpszToken, _T("InitReg"))==0)
    {
        return 0;
    }

    lpszToken = FindOneOf(lpszToken, szTokens);
}

// Are we Service or Local Server
CRegKey keyAppID;
LONG lRes = keyAppID.Open(HKEY_CLASSES_ROOT, _T("AppID"), KEY_READ);
if (lRes != ERROR_SUCCESS)
    return lRes;

CRegKey key;
lRes = key.Open(keyAppID, _T("{BF823564-E93B-11D3-B88C-CC792E000000}"), KEY_READ);
if (lRes != ERROR_SUCCESS)
    return lRes;

TCHAR szValue[_MAX_PATH];
DWORD dwLen = _MAX_PATH;
lRes = key.QueryValue(szValue, _T("LocalService"), &dwLen);

_Module.m_bService = FALSE;
if (lRes == ERROR_SUCCESS)
    _Module.m_bService = TRUE;

// AFX internal initialization
if (!AfxWinInit(hInstance, hPrevInstance, lpCmdLine, nShowCmd))
    CLogMsgEvent(LCEV_GENERIC, SVRTY_ERROR, "AfxWinInit failed.").Post(_logAll);
else
    _Module.Start();

// When we get here, the service has been stopped
return _Module.m_status.dwWin32ExitCode;
}
```

```
#include "stdafx.h"
#include "Logging.h"
#include "Registry.h"

////////////////////////////////////
////////////////////////////////////
// Log messages
////////////////////////////////////
////////////////////////////////////
CLogMsg::CLogMsg()
{
    m_pszText = NULL;
}

CLogMsg::CLogMsg(LPCSTR pszMessage)
{
    m_pszText = NULL;
    if (pszMessage != NULL)
        *this << pszMessage;
}

CLogMsg::CLogMsg(string & strMessage)
{
    m_pszText = NULL;
    *this << strMessage;
}

CLogMsg::~CLogMsg()
{
    ReleaseBuffers();
}

CLogMsg & CLogMsg::Format(LPCSTR pszFormat, ...)
{
    Clear();
    va_list pArgs;
    va_start(pArgs, pszFormat);
    TCHAR pszBuffer [1024];
    vsprintf(pszBuffer, pszFormat, pArgs);
    va_end(pArgs);
    *this << pszBuffer;
    return *this;
}

void CLogMsg::Post(CLogBase & log)
{
    log.Post(this);
    return;
}

long CLogMsg::Event()
{
    return 0;
}

long CLogMsg::Severity()
{
    return EVENTLOG_SUCCESS;
}

TCHAR ** CLogMsg::Arguments(long * plArgCount)
{
    *plArgCount = 1;
    Text();
    return &m_pszText;
}
```

```
TCHAR * CLogMsg::Text()
{
    ReleaseBuffers();
    *this << '\0';
    TCHAR * pszText = str();
    int nLen = pcount();
    m_pszText = new TCHAR [nLen + 1];
    _tcscpy(m_pszText, pszText);
    freeze(false);
    return m_pszText;
}

void CLogMsg::ReleaseBuffers()
{
    if (m_pszText != NULL)
    {
        delete [] m_pszText;
        m_pszText = NULL;
    }
    return;
}

void CLogMsg::Clear()
{
    ReleaseBuffers();
    seekp(0);
    return;
}

void CLogMsg::appendError(_com_error & e)
{
    string strError = (char *) e.Description();
    HRESULT hr = e.Error();
    *this << "COM Error = [" << strError << "]. hr = [" << std::hex << hr << "].";
    return;
}

void CLogMsg::appendError(HRESULT hr)
{
    *this << "hr = [" << std::hex << hr << std::dec << "].";
    return;
}

void CLogMsg::appendError(CLogMsg & em)
{
    appendError((std::stringstream &) em);
}

void CLogMsg::appendError(std::stringstream & strmError)
{
    strmError << '\0';
    *this << strmError.str();
    strmError.freeze(false);
}

void CLogMsg::setError(_com_error & e)
{
    clear();
    appendError(e);
}

void CLogMsg::setError(HRESULT hr)
{
    clear();
    appendError(hr);
}
```

```
void CLogMsg::setError(LPCSTR pszError)
{
    clear();
    *this << pszError;
}

void CLogMsg::setError(CLogMsg & em)
{
    clear();
    appendError(em);
}

void CLogMsg::getError(string & strError)
{
    *this << '\0';
    strError = str();
    freeze(false);
    return;
}

string CLogMsg::getError()
{
    string strError;
    *this << '\0';
    strError = str();
    freeze(false);
    return strError;
}

void CLogMsg::getError(std::stringstream & strmError)
{
    *this << '\0';
    strmError << str();
    freeze(false);
    return;
}

////////////////////////////////////

const char CLogMsgEvent::bArgSep = '\t';

CLogMsgEvent::CLogMsgEvent()
{
    Init();
}

CLogMsgEvent::CLogMsgEvent(LPCSTR pszMessage)
: CLogMsg(pszMessage)
{
    Init();
}

CLogMsgEvent::CLogMsgEvent(string & strMessage)
: CLogMsg(strMessage)
{
    Init();
}

CLogMsgEvent::CLogMsgEvent(long lEventID, long lSeverity, LPCSTR pszMessage)
: CLogMsg(pszMessage)
{
    Init();
    m_lEventID = lEventID;
    m_lSeverity = lSeverity;
}

CLogMsgEvent::CLogMsgEvent(long lEventID, long lSeverity, string & strMessage)
```

```
        :CLogMsg(strMessage)
    {
        Init();
        m_lEventID = lEventID;
        m_lSeverity = lSeverity;
    }

CLogMsgEvent::CLogMsgEvent(long lEventID, long lSeverity, __com_error & e)
{
    USES_CONVERSION;

    Init();
    m_lEventID = lEventID;
    m_lSeverity = lSeverity;

    *this << "0x" << std::hex << e.Error() << std::dec << bArgSep;
    BSTR bstrDesc = e.Description();
    if (bstrDesc != NULL)
        *this << W2T(bstrDesc);
    else
        *this << " ";
}

CLogMsgEvent::CLogMsgEvent(long lEventID, long lSeverity, HRESULT hr)
{
    Init();
    m_lEventID = lEventID;
    m_lSeverity = lSeverity;
    *this << "0x" << std::hex << hr;
}

CLogMsgEvent::~CLogMsgEvent()
{
    ReleaseBuffers();
}

inline void CLogMsgEvent::Init()
{
    m_lEventID = 0;
    m_lSeverity = -1;
    m_wArgCount = 0;
    m_ppszArgs = NULL;
}

void CLogMsgEvent::SetEvent(long lEventID, long lSeverity, LPCSTR pszMessage)
{
    Clear();
    m_lEventID = lEventID;
    m_lSeverity = lSeverity;
    if (pszMessage != NULL)
        *this << pszMessage;
}

long CLogMsgEvent::Event()
{
    return m_lEventID;
}

long CLogMsgEvent::Severity()
{
    if (m_lSeverity == -1)
    {
        if ((m_lEventID & 0xC0000000L) == 0xC0000000L)
            return EVENTLOG_ERROR_TYPE;
        else if (m_lEventID & 0x80000000L)
            return EVENTLOG_WARNING_TYPE;
        else if (m_lEventID & 0x40000000L)
```

```

        return EVENTLOG_INFORMATION_TYPE;
    else
        return EVENTLOG_SUCCESS;
    }
    else
        return m_lSeverity;
}

```

```

TCHAR ** CLogMsgEvent::Arguments(long * plArgCount)
{

```

```

    ReleaseBuffers();

    // get temp buffer
    stringstream strmTemp;
    *this << '\0';
    strmTemp << str();
    freeze(false);

    // make sure double nulled
    strmTemp << '\0' << '\0';

    TCHAR * pszText = strmTemp.str();
    if (*pszText)
        m_wArgCount++;

    // make array of strings
    for (int i = 0; pszText[i]; i++)
    {
        if (pszText[i] == CLogMsgEvent::bArgSep)
        {
            pszText[i] = 0;
            m_wArgCount++;
        }
    }

    // if data, allocate arg array
    if (m_wArgCount)
    {
        int nLen = 0;
        m_ppszArgs = new TCHAR * [m_wArgCount];
        for (int i = 0; i < m_wArgCount; i++)
        {
            nLen = _tcslen(pszText);
            m_ppszArgs[i] = new TCHAR [nLen + 1];
            _tcscpy(m_ppszArgs[i], pszText);
            pszText += nLen + 1;
        }
    }

    strmTemp.freeze(false);

    // return buffer
    *plArgCount = m_wArgCount;
    return m_ppszArgs;
}

```

```

TCHAR * CLogMsgEvent::Text()
{

```

```

    CLogMsg::ReleaseBuffers();

    // format message into temporary stringstream
    *this << '\0';
    std::stringstream strmTemp;
    strmTemp << "Event:0x" << std::hex << m_lEventID << ", Severity:" << std::dec <<
    Severity() << ", Text:";

    // if translation is turned on, then get message from message source

```



```
    DWORD dwCharsReturned = 0;
    if (CLogNTEvents::m_hMsgSrc != NULL)
    {
        TCHAR pszBuff [2048];
        dwCharsReturned = FormatMessage(FORMAT_MESSAGE_FROM_HMODULE |
        FORMAT_MESSAGE_ARGUMENT_ARRAY,
        CLogNTEvents::m_hMsgSrc,
        m_lEventID,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        pszBuff,
        2048,
        m_ppszArgs);

        if (dwCharsReturned)
        {
            // chop off line feed
            pszBuff[--dwCharsReturned] = 0;
            // move data to formatted message
            if (dwCharsReturned)
                strmTemp << pszBuff << '\0';
        }
    }

    // if translation not turned on or translation didn't work then put out argument data
    if (!dwCharsReturned)
    {
        strmTemp << str() << '\0';
        freeze(false);
    }

    // move temp stringstream into m_pszText and return pointer to m_pszText
    int nLength = strmTemp.pcount();
    m_pszText = new TCHAR [nLength + 1];
    _tcsncpy(m_pszText, strmTemp.str(), nLength);
    strmTemp.freeze(false);
    m_pszText[nLength] = 0;

    return m_pszText;
}

void CLogMsgEvent::ReleaseBuffers()
{
    CLogMsg::ReleaseBuffers();

    if (m_ppszArgs != NULL)
    {
        for (int i = 0; i < m_wArgCount; i++)
            delete [] m_ppszArgs[i];
        delete [] m_ppszArgs;
        m_ppszArgs = NULL;
        m_wArgCount = 0;
    }

    return;
}

////////////////////////////////////
////////////////////////////////////
// Logs
////////////////////////////////////
////////////////////////////////////

CLogBase::CLogBase()
{
    m_fEnabled = true;
    m_nIndent = 0;
}
```

```
CLogBase::CLogBase(LPCSTR pszResourceName)
{
    m_fEnabled = true;
    m_strResourceName = pszResourceName;
    m_nIndent = 0;
}

void CLogBase::ResourceName(LPCSTR pszResourceName)
{
    m_strResourceName = pszResourceName;
    return;
}

void CLogBase::Post(CLogMsg * pmsgLog)
{
    return;
}

void CLogBase::Open()
{
    return;
}

void CLogBase::Close()
{
    return;
}

////////////////////////////////////
HINSTANCE CLogNTEvents::m_hMsgSrc = NULL;

CLogNTEvents::CLogNTEvents()
    :CLogBase()
{
}

CLogNTEvents::CLogNTEvents(LPCSTR pszResourceName)
    :CLogBase(pszResourceName)
{
}

void CLogNTEvents::Post(CLogMsg * pmsgLog)
{
    if (!m_fEnabled)
        return;

    HANDLE hEventSource = RegisterEventSource(NULL, m_strResourceName.c_str());
    if (hEventSource != NULL)
    {
        long lArgCount;
        TCHAR ** pszArgs = pmsgLog->Arguments(&lArgCount);
        ReportEvent(hEventSource, pmsgLog->Severity(), 0, pmsgLog->Event(), NULL,
        lArgCount,
        0, (const TCHAR **) pszArgs, NULL);
        DeregisterEventSource(hEventSource);
    }
}

void CLogNTEvents::EnableTranslation(bool fEnable)
{
    if (fEnable)
    {
        if (!m_hMsgSrc)
            m_hMsgSrc = LoadMessageSource();
    }
    else
}
```

```

    {
        if (m_hMsgSrc)
        {
            FreeLibrary(m_hMsgSrc);
            m_hMsgSrc = NULL;
        }
    }

    return;
}

HINSTANCE CLogNTEvents::LoadMessageSource()
{
    CRegistry    regLocal;

    // get the name of the resource
    if (!regLocal.Connect(CRegistry::keyLocalMachine))
        return NULL;

    string strKey("SYSTEM\\CurrentControlSet\\Services\\EventLog\\Application\\");
    strKey += m_strResourceName;
    if (!regLocal.Open(strKey.c_str()))
        return NULL;

    string strDLL;
    if (!regLocal.GetValue("EventMessageFile", strDLL))
        return NULL;

    // load the library
    return LoadLibrary(strDLL.c_str());
}

////////////////////////////////////

CLogFile::CLogFile()
: CLogBase()
{
}

CLogFile::CLogFile(LPCSTR pszResourceName)
: CLogBase(pszResourceName)
{
    m_streamIO.open(pszResourceName, ios_base::out | ios_base::trunc);
}

void CLogFile::Open(LPCSTR pszFileName)
{
    Close();
    m_strResourceName = pszFileName;
    Open();
}

void CLogFile::Open()
{
    if (!m_streamIO.is_open())
        m_streamIO.open(m_strResourceName.c_str(), ios_base::out | ios_base::trunc);
}

void CLogFile::Close()
{
    if (m_streamIO.is_open())
        m_streamIO.close();
    return;
}

void CLogFile::Post(CLogMsg * pmsgLog)

```

```

{
    if (!m_fEnabled)
        return;

    Lock();
    if (m_streamIO.is_open())
    {
        string strTabs(m_nIndent, '\t');
        m_streamIO << strTabs << pmsgLog->Text() << '\n';
        m_streamIO.flush();
    }
    Unlock();

    return;
}

/////////////////////////////////////////////////////////////////

CLogDebug::CLogDebug()
{
}

void CLogDebug::Post(CLogMsg * pmsgLog)
{
    if (!m_fEnabled)
        return;
    if (m_nIndent)
    {
        string strTabs(m_nIndent, '\t');
        OutputDebugString(strTabs.c_str());
    }
    OutputDebugString(pmsgLog->Text());
    OutputDebugString("\n");
    return;
}

/////////////////////////////////////////////////////////////////

CLogMulti::CLogMulti()
{
}

void CLogMulti::AddLog(CLogBase * plog)
{
    Lock();
    m_collLogs.push_back(plog);
    Unlock();
}

void CLogMulti::RemoveLog(CLogBase * plog)
{
    Lock();
    if (plog != NULL)
        m_collLogs.remove(plog);
    else
        m_collLogs.erase(m_collLogs.begin(), m_collLogs.end());
    Unlock();
    return;
}

void CLogMulti::Post(CLogMsg * pmsgLog)
{
    if (!m_fEnabled)
        return;

    Lock();
    list<CLogBase *>::iterator    itLogs;

```

```
    for (itLogs = m_collLogs.begin(); itLogs != m_collLogs.end(); itLogs++)
        pmsgLog->Post>(*itLogs);
    Unlock();
    return;
}

void CLogMulti::Open()
{
    Lock();
    list<CLogBase *>::iterator itLogs;
    for (itLogs = m_collLogs.begin(); itLogs != m_collLogs.end(); itLogs++)
        (*itLogs)->Open();
    Unlock();
}

void CLogMulti::Close()
{
    Lock();
    list<CLogBase *>::iterator itLogs;
    for (itLogs = m_collLogs.begin(); itLogs != m_collLogs.end(); itLogs++)
        (*itLogs)->Close();
    Unlock();
}

string CTimeStamp::LocalTime()
{
    SYSTEMTIME tm;
    GetLocalTime(&tm);
    char pBuff [30];
    sprintf(pBuff, "%02d:%02d:%02d.%d", tm.wHour, tm.wMinute, tm.wSecond, tm.
        wMilliseconds);
    return string(pBuff);
}
```

```
#ifndef _Logging_h
#define _Logging_h
```

```
class CLogMsg;
```

```

////////////////////////////////////
////////////////////////////////////
log objects
////////////////////////////////////

```

```
Dependencies :
```

```

#include <string>
#include <list>
#include <fstream>
#include <strstream>
using namespace std;

```

```

////////////////////////////////////
////////////////////////////////////*/

```

```
class CLogBase
```

```

{
protected:
    bool                m_fEnabled;
    string              m_strResourceName;
    CComAutoCriticalSection m_syncCS;
    int                 m_nIndent;

```

```
public:
```

```

    CLogBase();
    CLogBase(LPCSTR pszResourceName);
    void Enabled(bool fEnabled){m_fEnabled = fEnabled;}
    bool Enabled(){return m_fEnabled;}
    void Lock(){m_syncCS.Lock();}
    void Unlock(){m_syncCS.Unlock();}
    virtual void Post(CLogMsg * pmsgLog) = 0;
    virtual void ResourceName(LPCSTR pszResourceName);
    void GetResourceName(string & strResourceName){strResourceName = m_strResourceName;}
    virtual void Open();
    virtual void Close();
    void PushIndent(){m_nIndent++;}
    void PopIndent(){if (m_nIndent > 0) m_nIndent--;}

```

```
};
```

```

////////////////////////////////////

```

```
class CLogNTEvents : public CLogBase
```

```

{
    friend class CLogMsgEvent;

protected:
    static HINSTANCE      m_hMsgSrc;

```

```
    HINSTANCE LoadMessageSource();
```

```
public:
```

```

    CLogNTEvents();
    CLogNTEvents(LPCSTR pszResourceName);
    virtual void Post(CLogMsg * pmsgLog);
    void EnableTranslation(bool fEnable);

```

```
};
```

```

////////////////////////////////////

```

```
class CLogFile : public CLogBase
```

```

{
protected:
    fstream          m_streamIO;

public:
    CLogFile();
    CLogFile(LPCSTR pszResourceName);
    void Open(LPCSTR pszFileName);
    virtual void Post(CLogMsg * pmsgLog);
    virtual void Open();
    virtual void Close();
};

////////////////////////////////////

class CLogDebug : public CLogBase
{
public:
    CLogDebug();
    virtual void Post(CLogMsg * pmsgLog);
};

////////////////////////////////////

class CLogMulti : public CLogBase
{
protected:
    list<CLogBase *>    m_collLogs;

public:
    CLogMulti();
    void AddLog(CLogBase * plog);
    void RemoveLog(CLogBase * plog);
    virtual void Post(CLogMsg * pmsgLog);
    virtual void Open();
    virtual void Close();
};

////////////////////////////////////
////////////////////////////////////
// message objects
////////////////////////////////////
////////////////////////////////////

class CLogMsg : public std::stringstream
{
protected:
    TCHAR *          m_pszText;
    virtual void ReleaseBuffers();

public:
    CLogMsg();
    CLogMsg(LPCSTR pszMessage);
    CLogMsg(string & strMessage);
    virtual ~CLogMsg();
    CLogMsg & Format(LPCSTR pszFormat, ...);
    virtual void Post(CLogBase & log);
    virtual long Event();
    virtual long Severity();
    virtual TCHAR ** Arguments(long * plArgCount);
    virtual TCHAR * Text();
    virtual void Clear();
    void appendError(_com_error & e);
    void appendError(HRESULT hr);
    void appendError(CLogMsg & em);
    void appendError(std::stringstream & strmError);
    void setError(_com_error & e);
};

```

```

    void setError(HRESULT hr);
    void setError(LPCSTR pszError);
    void setError(CLogMsg & em);
    void getError(string & strError);
    string getError();
    void getError(std::stringstream & strmError);
    void clear(){Clear();}
};

////////////////////////////////////

enum { SVRTY_DEFAULT    = -1,
        SVRTY_SUCCESS  = EVENTLOG_SUCCESS,
        SVRTY_ERROR    = EVENTLOG_ERROR_TYPE,
        SVRTY_WARNING  = EVENTLOG_WARNING_TYPE,
        SVRTY_INFO     = EVENTLOG_INFORMATION_TYPE };

class CLogMsgEvent : public CLogMsg
{
protected:
    long          m_lEventID;
    long          m_lSeverity;
    short         m_wArgCount;
    TCHAR **      m_ppszArgs;

    void Init();
    virtual void ReleaseBuffers();

public:
    static const char    bArgSep;

public:
    CLogMsgEvent();
    CLogMsgEvent(LPCSTR pszMessage);
    CLogMsgEvent(string & strMessage);
    CLogMsgEvent(long lEventID, long lSeverity = -1, LPCSTR pszMessage = NULL);
    CLogMsgEvent(long lEventID, long lSeverity, string & strMessage);
    CLogMsgEvent(long lEventID, long lSeverity, _com_error & e);
    CLogMsgEvent(long lEventID, long lSeverity, HRESULT hr);
    ~CLogMsgEvent();
    void SetEvent(long lEventID, long lSeverity = -1, LPCSTR pszMessage = NULL);
    virtual long Event();
    virtual long Severity();
    virtual TCHAR ** Arguments(long * plArgCount);
    virtual TCHAR * Text();
};

class CTimeStamp
{
public:
    static string LocalTime();
};

#endif

```



```

    {
        _recursively_delete_all_sub_keys( child_key_handle, temporary_key_name );

        return_value = RegEnumKey( child_key_handle, 0, temporary_key_name, MAX_PATH );
    }

    delete [] temporary_key_name;
    temporary_key_name = NULL;
    RegCloseKey( child_key_handle );
    return_value = RegDeleteKey( key_handle, key_name );
    return( return_value );
}

/*****
Function name: CRegistry::CRegistry
Description   :
Return type   :
*****/
CRegistry::CRegistry()
{
    m_Initialize();
}

/*****
Function name: CRegistry::~~CRegistry
Description   :
Return type   :
*****/
CRegistry::~~CRegistry()
{
    if ( m_RegistryHandle != (HKEY) NULL )
    {
        Close();
    }

    m_Initialize();
}

/*****
Function name: CRegistry::m_Initialize
Description   :
Return type   : void
Argument      : void
*****/
void CRegistry::m_Initialize( void )
{
    _ASSERT( this );

    /*
    ** Make sure everything is zeroed out
    */

    m_ClassName.erase();
    m_ComputerName.erase();
    m_KeyName.erase();
    m_RegistryName.erase();

    m_KeyHandle           = (HKEY) NULL;
    m_ErrorCode           = 0L;
    m_NumberOfSubkeys     = 0;
    m_LongestSubkeyNameLength = 0;
    m_LongestClassNameLength = 0;
    m_NumberOfValues      = 0;
    m_LongestValueNameLength = 0;
    m_LongestValueDataLength = 0;
    m_SecurityDescriptorLength = 0;
    m_LastWriteTime.dwLowDateTime = 0;

```

```

    m_LastWriteTime.dwHighDateTime = 0;
    m_RegistryHandle = (HKEY) NULL;
}

/*****
Function name: CRegistry::Close
Description   :
Return type  : BOOL
Argument     : void
*****/
BOOL CRegistry::Close( void )
{
    _ASSERT( this );

    if ( m_KeyHandle != (HKEY) NULL )
    {
        ::RegCloseKey( m_KeyHandle );
        m_KeyHandle = (HKEY) NULL;
    }

    if ( m_RegistryHandle == (HKEY) NULL )
    {
        return( TRUE );
    }

    m_ErrorCode = ::RegCloseKey( m_RegistryHandle );

    if ( m_ErrorCode == ERROR_SUCCESS )
    {
        m_RegistryHandle = (HKEY) NULL;
        m_Initialize();

        return( TRUE );
    }
    else
    {
        return( FALSE );
    }
}

/*****
Function name: CRegistry::Connect
Description   :
Return type  : BOOL
Argument     : HKEY key_to_open
Argument     : LPCTSTR name_of_computer
*****/
BOOL CRegistry::Connect( const _Keys key_to_open, LPCTSTR name_of_computer )
{
    _ASSERT( this );

    // We were passed a pointer, do not trust it
    try
    {
        /*
        ** name_of_computer can be NULL
        */

        if ( key_to_open == keyClassesRoot || key_to_open == keyCurrentUser )
        {
            if ( name_of_computer == NULL )
            {
                m_RegistryHandle = (HKEY)key_to_open;
                m_ErrorCode = ERROR_SUCCESS;
            }
            else
            {

```

```

    /*
    ** NT won't allow you to connect to these hives via RegConnectRegistry so we
    'll just skip that step
    */
    m_ErrorCode = ERROR_INVALID_HANDLE;
}
else
{
    // Thanks to Paul Ostrowski [postrowski@xantel.com] for finding UNICODE bug here
    // RegConnectRegistry is not const correct
    m_ErrorCode = ::RegConnectRegistry( (LPTSTR) name_of_computer, (HKEY)key_to_open,
    &m_RegistryHandle );

    if ( m_ErrorCode == ERROR_SUCCESS )
    {
        if ( name_of_computer == NULL )
        {
            TCHAR computer_name[ MAX_PATH ];
            DWORD size = MAX_PATH;

            if ( ::GetComputerName( computer_name, &size ) == FALSE )
            {
                m_ComputerName.erase();
            }
            else
            {
                m_ComputerName = computer_name;
            }
        }
        else
        {
            m_ComputerName = name_of_computer;
        }

        //
        // It would be nice to use a switch statement here but I get a "not integral"
        error!
        //

        if ( (HKEY)key_to_open == HKEY_LOCAL_MACHINE )
        {
            m_RegistryName = TEXT( "HKEY_LOCAL_MACHINE" );
        }
        else if ( (HKEY)key_to_open == HKEY_CLASSES_ROOT )
        {
            m_RegistryName = TEXT( "HKEY_CLASSES_ROOT" );
        }
        else if ( (HKEY)key_to_open == HKEY_USERS )
        {
            m_RegistryName = TEXT( "HKEY_USERS" );
        }
        else if ( (HKEY)key_to_open == HKEY_CURRENT_USER )
        {
            m_RegistryName = TEXT( "HKEY_CURRENT_USER" );
        }
        else if ( (HKEY)key_to_open == HKEY_PERFORMANCE_DATA )
        {
            m_RegistryName = TEXT( "HKEY_PERFORMANCE_DATA" );
        }
    }
    #if ( WINVER >= 0x400 )
        else if ( (HKEY)key_to_open == HKEY_CURRENT_CONFIG )
        {
            m_RegistryName = TEXT( "HKEY_CURRENT_CONFIG" );
        }
        else if ( (HKEY)key_to_open == HKEY_DYN_DATA )
    
```

```

    {
        m_RegistryName = TEXT( "HKEY_DYN_DATA" );
    }
#endif
    else
    {
        m_RegistryName = TEXT( "Unknown" );
    }

    return( TRUE );
}
else
{
    return( FALSE );
}
}
catch( ... )
{
    m_ErrorCode = ERROR_EXCEPTION_IN_SERVICE;
    return( FALSE );
}
}

/*****
Function name: CRegistry::Create
Description   :
Return type   : BOOL
Argument      : LPCTSTR          name_of_subkey
Argument      : LPCTSTR          name_of_class
Argument      : CreateOptions     options
Argument      : CreatePermissions permissions
Argument      : LPSECURITY_ATTRIBUTES security_attributes_p
Argument      : CreationDisposition * disposition_p
*****/
BOOL CRegistry::Create( LPCTSTR          name_of_subkey,
                       LPCTSTR          name_of_class,
                       CreateOptions     options,
                       CreatePermissions permissions,
                       LPSECURITY_ATTRIBUTES security_attributes_p,
                       CreationDisposition * disposition_p )
{
    _ASSERT( this );
    _ASSERT( name_of_subkey != NULL );

    if ( name_of_subkey == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    // We were passed a pointer, do not trust it
    try
    {
        DWORD disposition = 0;

        if ( name_of_class == NULL )
        {
            name_of_class = TEXT( "" ); // Paul Ostrowski [postrowski@xantel.com]
        }

        if ( m_KeyHandle != (HKEY) NULL )
        {
            ::RegCloseKey( m_KeyHandle );
            m_KeyHandle = (HKEY) NULL;
        }

        m_ErrorCode = ::RegCreateKeyEx( m_RegistryHandle,

```

```

        name_of_subkey,
        (DWORD) 0,
        (LPTSTR) name_of_class, // Paul Ostrowski
    [postrowski@zantel.com]
        options,
        permissions,
        security_attributes_p,
        &m_KeyHandle,
        &disposition );

    if ( m_ErrorCode == ERROR_SUCCESS )
    {
        if ( disposition_p != NULL )
        {
            *disposition_p = (CreationDisposition) disposition;
        }

        m_KeyName = name_of_subkey;

        return( TRUE );
    }
    else
    {
        return( FALSE );
    }
}
catch( ... )
{
    m_ErrorCode = ERROR_EXCEPTION_IN_SERVICE;
    return( FALSE );
}
}

/******
Function name: CRegistry::DeleteKey
Description   :
Return type   : BOOL
Argument      : LPCTSTR name_of_key_to_delete
*****
BOOL CRegistry::DeleteKey( LPCTSTR name_of_key_to_delete )
{
    _ASSERT( this );
    _ASSERT( name_of_key_to_delete != NULL );

    if ( name_of_key_to_delete == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    // We were passed a pointer, do not trust it
    try
    {
        /*
        ** You can't delete a key given a full path. What you have to do is back up one
        level and then do a delete
        */

        string full_key_name = name_of_key_to_delete;

        if ( full_key_name.find( TEXT( '\\\' ) ) == (-1) )
        {
            /*
            ** User had not given us a full path so assume the name of the key he passed us
            ** is a key off of the current key
            */

```

```

        m_ErrorMessage = ::_recursively_delete_all_sub_keys( m_KeyHandle,
name_of_key_to_delete );
    }
    else
    {
        int last_back_slash_location = full_key_name.size() - 1;

        /*
        ** We know this loop will succeed because a back slash was found in the above if
statement
        */

        while( full_key_name[ last_back_slash_location ] != TEXT( '\\' ) )
        {
            last_back_slash_location--;
        }

        string currently_opened_key_name = m_KeyName;

        string parent_key_name = full_key_name.substr( 0, last_back_slash_location );
        int nCount = ( full_key_name.size() - last_back_slash_location ) - 1;
        string child_key_name = full_key_name.substr(full_key_name.size() - nCount,
nCount);

        /*
        ** Now we open the parent key and delete the child
        */

        if ( Open( parent_key_name.c_str() ) != FALSE )
        {
            m_ErrorMessage = ::_recursively_delete_all_sub_keys( m_KeyHandle, child_key_name.
c_str() );
        }
        else
        {
            {
                m_KeyName = currently_opened_key_name;
                return( FALSE );
            }
        }

        if ( m_ErrorMessage == ERROR_SUCCESS )
        {
            return( TRUE );
        }
        else
        {
            return( FALSE );
        }
    }
}
catch( ... )
{
    m_ErrorMessage = ERROR_EXCEPTION_IN_SERVICE;
    return( FALSE );
}
}

/*****
Function name: CRegistry::DeleteValue
Description :
Return type : BOOL
Argument : LPCTSTR name_of_value_to_delete
*****/
BOOL CRegistry::DeleteValue( LPCTSTR name_of_value_to_delete )
{
    _ASSERT( this );

    /*

```

```

    ** name_of_value_to_delete can be NULL
    */

    // We were passed a pointer, do not trust it
    try
    {
        m_ErrorCode = ::RegDeleteValue( m_KeyHandle, (LPTSTR) name_of_value_to_delete );

        if ( m_ErrorCode == ERROR_SUCCESS )
        {
            return( TRUE );
        }
        else
        {
            return( FALSE );
        }
    }
    catch( ... )
    {
        m_ErrorCode = ERROR_EXCEPTION_IN_SERVICE;
        return( FALSE );
    }
}

/*****
Function name: CRegistry::EnumerateKeys
Description :
Return type : BOOL
Argument : const DWORD subkey_index
Argument : string& subkey_name
Argument : string& class_name
*****/
BOOL CRegistry::EnumerateKeys( const DWORD subkey_index, string& subkey_name, string&
    class_name )
{
    _ASSERTE( this );

    TCHAR subkey_name_string[ 2048 ];
    TCHAR class_name_string[ 2048 ];

    DWORD size_of_subkey_name_string = (sizeof(subkey_name_string)/sizeof(*
        (subkey_name_string))) - 1;
    DWORD size_of_class_name_string = (sizeof(class_name_string)/sizeof(*
        (class_name_string))) - 1;

    ::ZeroMemory( subkey_name_string, sizeof( subkey_name_string ) );
    ::ZeroMemory( class_name_string, sizeof( class_name_string ) );

    m_ErrorCode = ::RegEnumKeyEx( m_KeyHandle,
        subkey_index,
        subkey_name_string,
        &size_of_subkey_name_string,
        NULL,
        class_name_string,
        &size_of_class_name_string,
        &m_LastWriteTime );

    if ( m_ErrorCode == ERROR_SUCCESS )
    {
        subkey_name = subkey_name_string;
        class_name = class_name_string;
        return( TRUE );
    }
    else
    {
        return( FALSE );
    }
}

```



```

)

/*****
Function name: CRegistry::EnumerateValues
Description   :
Return type   : BOOL
Argument      : const DWORD    value_index
Argument      : string&        name_of_value
Argument      : KeyValueTypes& type_code
Argument      : LPBYTE         data_buffer
Argument      : DWORD&         size_of_data_buffer
*****/
BOOL CRegistry::EnumerateValues( const DWORD    value_index,
                                string&        name_of_value,
                                KeyValueTypes& type_code,
                                LPBYTE         data_buffer,
                                DWORD&         size_of_data_buffer )
{
    _ASSERT( this );

    /*
    ** data_buffer and size_of_data_buffer can be NULL
    */

    DWORD temp_type_code = type_code;

    TCHAR temp_name[ 2048 ];

    ::ZeroMemory( temp_name, sizeof( temp_name ) );
    DWORD temp_name_size = (sizeof(temp_name)/sizeof(*(temp_name)));

    // We were passed a pointer, do not trust it
    try
    {
        m_ErrorCode = ::RegEnumValue( m_KeyHandle,
                                      value_index,
                                      temp_name,
                                      &temp_name_size,
                                      NULL,
                                      &temp_type_code,
                                      data_buffer,
                                      &size_of_data_buffer );

        if ( m_ErrorCode == ERROR_SUCCESS )
        {
            type_code      = (KeyValueTypes) temp_type_code;
            name_of_value = temp_name;

            return( TRUE );
        }
        else
        {
            return( FALSE );
        }
    }
    catch( ... )
    {
        m_ErrorCode = ERROR_EXCEPTION_IN_SERVICE;
        return( FALSE );
    }
}

/*****
Function name: CRegistry::Flush
Description   :
Return type   : BOOL
Argument      : void
*****/

```



```

        return_value = TRUE;
    }
    else
    {
        return_value = FALSE;
    }

    delete [] memory_buffer;
    return( return_value );
}

/*****
Function name: CRegistry::GetClassName
Description   :
Return type  : void
Argument     : string& class_name
*****/
void CRegistry::GetClassName( string& class_name ) const
{
    class_name = m_ClassName;
}

/*****
Function name: CRegistry::GetComputerName
Description   :
Return type  : void
Argument     : string& computer_name
*****/
void CRegistry::GetComputerName( string& computer_name ) const
{
    computer_name = m_ComputerName;
}

/*****
Function name: CRegistry::GetDoubleWordValue
Description   :
Return type  : BOOL
Argument     : LPCTSTR name_of_value
Argument     : DWORD& return_value
*****/
BOOL CRegistry::GetDoubleWordValue( LPCTSTR name_of_value, DWORD& return_value )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    DWORD size_of_buffer = sizeof( DWORD );
    KeyValueTypes type = typeDoubleWord;
    return( QueryValue( name_of_value, type, (LPBYTE) &return_value, size_of_buffer ) );
}

/*****
Function name: CRegistry::GetErrorCode
Description   :
Return type  : BOOL
Argument     : void
*****/
BOOL CRegistry::GetErrorCode( void ) const
{
    _ASSERT( this );
    return( m_ErrorCode );
}

```

```

/*****
Function name: CRegistry::GetKeyName
Description   :
Return type  : void
Argument     : string& key_name
*****/
void CRegistry::GetKeyName( string& key_name ) const
{
    key_name = m_KeyName;
}

/*****
Function name: CRegistry::GetNumberOfSubkeys
Description   :
Return type   : DWORD
Argument      : void
*****/
DWORD CRegistry::GetNumberOfSubkeys( void ) const
{
    return( m_NumberOfSubkeys );
}

/*****
Function name: CRegistry::GetNumberOfValues
Description   :
Return type   : DWORD
Argument      : void
*****/
DWORD CRegistry::GetNumberOfValues( void ) const
{
    return( m_NumberOfValues );
}

/*****
Function name: CRegistry::GetRegistryName
Description   :
Return type   : void
Argument      : string& registry_name
*****/
void CRegistry::GetRegistryName( string& registry_name ) const
{
    registry_name = m_RegistryName;
}

/*****
Function name: CRegistry::GetStringValue
Description   :
Return type   : BOOL
Argument      : LPCTSTR name_of_value
Argument      : string& return_string
*****/
BOOL CRegistry::GetStringValue( LPCTSTR name_of_value, string& return_string )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    TCHAR temp_string[ 2048 ];
    DWORD size_of_buffer = 2048;

    ::ZeroMemory( temp_string, sizeof( temp_string ) );

```

```

    KeyValueTypes type = typeString;

    if ( QueryValue( name_of_value, type, (LPBYTE) temp_string, size_of_buffer ) != FALSE )
    {
        return_string = temp_string;
        return( TRUE );
    }
    else
    {
        return_string.erase();
        return( FALSE );
    }
}

/* *****
Function name: CRegistry::GetValue
Description   :
Return type   : BOOL
Argument      : LPCTSTR name_of_value
Argument      : DWORD& return_value
***** */
BOOL CRegistry::GetValue( LPCTSTR name_of_value, DWORD& return_value )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    return( GetDoubleWordValue( name_of_value, return_value ) );
}

/* *****
Function name: CRegistry::GetValue
Description   :
Return type   : BOOL
Argument      : LPCTSTR name_of_value
Argument      : string& return_string
***** */
BOOL CRegistry::GetValue( LPCTSTR name_of_value, string& return_string )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    return( GetStringValue( name_of_value, return_string ) );
}

/* *****
Function name: CRegistry::Open
Description   :
Return type   : BOOL
Argument      : LPCTSTR name_of_subkey_to_open
Argument      : const CreatePermissions security_access_mask
***** */
BOOL CRegistry::Open( LPCTSTR name_of_subkey_to_open, const CreatePermissions
    security_access_mask )
{

```

```

    _ASSERT( this );

    /*
    ** name_of_subkey_to_open can be NULL
    */

    // We were passed a pointer, do not trust it
    try
    {
        if ( m_KeyHandle != (HKEY) NULL )
        {
            ::RegCloseKey( m_KeyHandle );
            m_KeyHandle = (HKEY) NULL;
        }

        m_ErrorCode = ::RegOpenKeyEx( m_RegistryHandle, name_of_subkey_to_open, NULL,
        security_access_mask, &m_KeyHandle );

        if ( m_ErrorCode == ERROR_SUCCESS )
        {
            QueryInfo();
            m_KeyName = name_of_subkey_to_open;

            return( TRUE );
        }
        else
        {
            return( FALSE );
        }
    }
    catch( ... )
    {
        m_ErrorCode = ERROR_EXCEPTION_IN_SERVICE;
        return( FALSE );
    }
}

/*****
Function name: CRegistry::QueryInfo
Description   :
Return type   : BOOL
Argument      : void
*****/
BOOL CRegistry::QueryInfo( void )
{
    _ASSERT( this );
    TCHAR class_name[ 2048 ];
    ::ZeroMemory( class_name, sizeof( class_name ) );
    DWORD size_of_class_name = (sizeof(class_name)/sizeof(*(class_name))) - 1;

    m_ErrorCode = ::RegQueryInfoKey( m_KeyHandle,
                                    class_name,
                                    &size_of_class_name,
                                    (LPDWORD) NULL,
                                    &m_NumberOfSubkeys,
                                    &m_LongestSubkeyNameLength,
                                    &m_LongestClassNameLength,
                                    &m_NumberOfValues,
                                    &m_LongestValueNameLength,
                                    &m_LongestValueDataLength,
                                    &m_SecurityDescriptorLength,
                                    &m_LastWriteTime );

    if ( m_ErrorCode == ERROR_SUCCESS )
    {
        m_ClassName = class_name;
        return( TRUE );
    }
}

```

```

    }
    else
    {
        return( FALSE );
    }
}

```

```

/*****

```

```

Function name: CRegistry::QueryValue
Description   :
Return type   : BOOL
Argument      : LPCTSTR          name_of_value
Argument      : KeyValueTypes&  value_type
Argument      : LPBYTE          address_of_buffer
Argument      : DWORD&          size_of_buffer
*****/

```

```

BOOL CRegistry::QueryValue( LPCTSTR          name_of_value,
                           KeyValueTypes&  value_type,
                           LPBYTE          address_of_buffer,
                           DWORD&          size_of_buffer )

```

```

{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    /*
    ** address_of_buffer and size_of_buffer can be NULL
    */

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    // We were passed a pointer, do not trust it

    try
    {
        DWORD temp_data_type = (DWORD) value_type;

        m_ErrorCode = ::RegQueryValueEx( m_KeyHandle,
                                          (LPTSTR) name_of_value,
                                          NULL,
                                          &temp_data_type,
                                          address_of_buffer,
                                          &size_of_buffer );

        if ( m_ErrorCode == ERROR_SUCCESS )
        {
            value_type = (KeyValueTypes) temp_data_type;
            return( TRUE );
        }
        else
        {
            return( FALSE );
        }
    }
    catch( ... )
    {
        m_ErrorCode = ERROR_EXCEPTION_IN_SERVICE;
        return( FALSE );
    }
}

```

```

/*****
Function name: CRegistry::SetBinaryValue

```

```

Description :
Return type : BOOL
Argument    : LPCTSTR name_of_value
Argument    : const BYTE bytes_to_write[]
Argument    : DWORD num_bytes_to_write

```

```

*****
BOOL CRegistry::SetBinaryValue( LPCTSTR name_of_value, const BYTE bytes_to_write[], DWORD
num_bytes_to_write )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    BOOL return_value = SetValue( name_of_value, typeBinary, (LPBYTE)bytes_to_write,
num_bytes_to_write );
    return( return_value );
}

```

```

Function name: CRegistry::SetDoubleWordValue
Description :
Return type : BOOL
Argument    : LPCTSTR name_of_value
Argument    : DWORD value_to_write

```

```

*****
BOOL CRegistry::SetDoubleWordValue( LPCTSTR name_of_value, DWORD value_to_write )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    return( SetValue( name_of_value, typeDoubleWord, (const PBYTE) &value_to_write, sizeof(
DWORD ) ) );
}

```

```

Function name: CRegistry::SetStringValue
Description :
Return type : BOOL
Argument    : LPCTSTR name_of_value
Argument    : const string& string_value

```

```

*****
BOOL CRegistry::SetStringValue( LPCTSTR name_of_value, const string& string_value )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    return( SetValue( name_of_value, typeString, (const PBYTE) string_value.c_str(),
string_value.size() + 1 ) );
}

```



```

/*****
Function name: CRegistry::SetValue
Description   :
Return type  : BOOL
Argument     : LPCTSTR name_of_value
Argument     : DWORD value
*****/
BOOL CRegistry::SetValue( LPCTSTR name_of_value, DWORD value )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    return( SetDoubleWordValue( name_of_value, value ) );
}

/*****
Function name: CRegistry::SetValue
Description   :
Return type  : BOOL
Argument     : LPCTSTR name_of_value
Argument     : const string& string_to_write
*****/
BOOL CRegistry::SetValue( LPCTSTR name_of_value, const string& string_to_write )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );

    if ( name_of_value == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }

    return( SetStringValue( name_of_value, string_to_write ) );
}

/*****
Function name: CRegistry::SetValue
Description   :
Return type  : BOOL
Argument     : LPCTSTR name_of_value
Argument     : const KeyValueType type_of_value_to_set
Argument     : const PBYTE address_of_value_data
Argument     : const DWORD size_of_data
*****/
BOOL CRegistry::SetValue( LPCTSTR name_of_value,
                          const KeyValueType type_of_value_to_set,
                          const PBYTE address_of_value_data,
                          const DWORD size_of_data )
{
    _ASSERT( this );
    _ASSERT( name_of_value != NULL );
    _ASSERT( address_of_value_data != NULL );

    if ( name_of_value == NULL || address_of_value_data == NULL )
    {
        m_ErrorCode = ERROR_INVALID_PARAMETER;
        return( FALSE );
    }
}

```

```
// We were passed a pointer, do not trust it

try
{
    m_ErrorCode = ::RegSetValueEx( m_KeyHandle,
                                    name_of_value,
                                    0,
                                    type_of_value_to_set,
                                    address_of_value_data,
                                    size_of_data );

    if ( m_ErrorCode == ERROR_SUCCESS )
    {
        return( TRUE );
    }
    else
    {
        return( FALSE );
    }
}
catch( ... )
{
    m_ErrorCode = ERROR_EXCEPTION_IN_SERVICE;
    return( FALSE );
}
}
```

```

#include "stdafx.h"
#include "registryBase.h"
// CRegistryBase

CRegistryBase::CRegistryBase()
{
}

bool CRegistryBase::connect(CRegistry::_Keys eKey, LPCSTR pszComputer)
{
    if (m_oRegistry.Connect(eKey) == REG_FAILURE)
    {
        string strRegName;
        m_oRegistry.GetRegistryName(strRegName);

        m_strLastError = "Unable to connect to [";
        m_strLastError += strRegName;
        m_strLastError += "]";
        if (pszComputer != NULL)
        {
            m_strLastError += " on computer [";
            m_strLastError += pszComputer;
            m_strLastError += "]";
        }
        return false;
    }

    return true;
}

bool CRegistryBase::getValue(LPCSTR pszValueName, string & strValue)
{
    if (m_oRegistry.GetValue(pszValueName, strValue) == REG_FAILURE)
    {
        m_strLastError = "Unable to retrieve value [";
        m_strLastError += pszValueName;
        m_strLastError += "]";
        return false;
    }

    return true;
}

bool CRegistryBase::getValue(LPCSTR pszValueName, unsigned long & lValue)
{
    if (m_oRegistry.GetValue(pszValueName, lValue) == REG_FAILURE)
    {
        m_strLastError = "Unable to retrieve value [";
        m_strLastError += pszValueName;
        m_strLastError += "]";
        return false;
    }

    return true;
}

bool CRegistryBase::getBinaryValue(LPCSTR pszValueName, LPBYTE pBuff, DWORD * pdwBuffSize)
{
    CRegistry::_KeyValueTypes eValueTypes = CRegistry::_typeBinary;
    if (m_oRegistry.QueryValue(pszValueName, eValueTypes, pBuff,
        *pdwBuffSize) == REG_FAILURE)
    {
        m_strLastError = "Unable to retrieve value [";
        m_strLastError += pszValueName;
        m_strLastError += "]";
        return false;
    }

    return true;
}

```

```
}

int CRegistryBase::enumKeys(vector<string> & aryKeys)
{
    DWORD dwIdx = 0;
    string strKeyName;
    string strClassName;
    while (m_oRegistry.EnumerateKeys(dwIdx++, strKeyName, strClassName))
        aryKeys.push_back(strKeyName);
    return aryKeys.size();
}

bool CRegistryBase::setValue(LPCSTR pszValueName, LPCSTR pszValue)
{
    if (m_oRegistry.SetValue(pszValueName, CRegistry::typeString, (PBYTE) pszValue,
        strlen(pszValue) + 1) == REG_FAILURE)
    {
        m_strLastError = "Unable to write value [";
        m_strLastError += pszValueName;
        m_strLastError += "]\n";
        return false;
    }

    return true;
}

bool CRegistryBase::setValue(LPCSTR pszValueName, unsigned long lValue)
{
    if (m_oRegistry.SetValue(pszValueName, CRegistry::typeDoubleWord, (PBYTE) &lValue,
        sizeof(lValue)) == REG_FAILURE)
    {
        m_strLastError = "Unable to write value [";
        m_strLastError += pszValueName;
        m_strLastError += "]\n";
        return false;
    }

    return true;
}

bool CRegistryBase::openKey(LPCSTR pszKeyPath)
{
    if (m_oRegistry.Open(pszKeyPath, CRegistry::permissionAllAccess) == REG_FAILURE)
    {
        m_strLastError = "Unable to open key [";
        m_strLastError += pszKeyPath;
        m_strLastError += "]\n";
        return false;
    }

    return true;
}

bool CRegistryBase::createKey(LPCSTR pszKeyPath)
{
    CRegistry::CreationDisposition eDisposition;

    if (m_oRegistry.Create(pszKeyPath,
        NULL,
        CRegistry::optionsNonVolatile,
        CRegistry::permissionAllAccess,
        NULL,
        &eDisposition) == REG_FAILURE)
    {
        m_strLastError = "Unable to create key [";
        m_strLastError += pszKeyPath;
        m_strLastError += "]\n";
        return false;
    }
}
```

```
    }  
    return true;  
}
```

```
#ifndef _registryBase_h
#define _registryBase_h

#include "registry.h"

class CRegistryBase
{
protected:
    CRegistry          m_oRegistry;

    enum {REG_FAILURE = 0, REG_SUCCESS = 1};

    CRegistryBase();
    bool connect(CRegistry::_Keys eKey, LPCSTR pszComputer = NULL);
    bool getValue(LPCSTR pszValueName, string & strValue);
    bool getValue(LPCSTR pszValueName, unsigned long & lValue);
    bool getBinaryValue(LPCSTR pszValueName, LPBYTE pBuff, DWORD * pdwBuffSize);
    bool setValue(LPCSTR pszValueName, LPCSTR pszValue);
    bool setValue(LPCSTR pszValueName, unsigned long lValue);
    bool openKey(LPCSTR pszKeyPath);
    bool createKey(LPCSTR pszKeyPath);
    int enumKeys(vector<string> & aryKeys);
    bool close() { return (m_oRegistry.Close() == TRUE); }

public:
    string          m_strLastError;
};

#endif
```

```
/**(NO_DEPENDENCIES)*/
// Microsoft Visual C++ generated include file.
// Used by LCKioskServer.rc
//
#define IDS_SERVICENAME 100

#define IDR_LCKioskServer 100

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifdef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE 201
#define _APS_NEXT_COMMAND_VALUE 32768
#define _APS_NEXT_CONTROL_VALUE 201
#define _APS_NEXT_SYMED_VALUE 101
#endif
#endif
```

```
// stdafx.cpp : source file that includes just the standard includes
//  stdafx.pch will be the pre-compiled header
//  stdafx.obj will contain the pre-compiled type information
```

```
#include "stdafx.h"
```

```
#ifdef _ATL_STATIC_REGISTRY
#include <statreg.h>
#include <statreg.cpp>
#endif
```

```
#include <atlimpl.cpp>
```



```
// stdafx.h : include file for standard system include files,
//          or project specific include files that are used frequently,
//          but are changed infrequently

#if !defined(AFX_STDAFX_H__BF823566_E93B_11D3_B88C_CC792E000000__INCLUDED_)
#define AFX_STDAFX_H__BF823566_E93B_11D3_B88C_CC792E000000__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define _WIN32_DCOM
#define STRICT

#include "afxdisp.h"          // MFC support
#include "afxinet.h"          // MFC internet classes
#include "afxmt.h"            // MFC thread synchronization

#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0400
#endif

// #define _ATL_APARTMENT_THREADED

#include <atlbase.h>
// You may derive a class from CComModule and use it if you want to override
// something, but do not change the name of _Module

class CServiceModule : public CComModule
{
public:
    HRESULT RegisterServer(BOOL bRegTypeLib, BOOL bService);
    HRESULT UnregisterServer();
    void Init(_ATL_OBJMAP_ENTRY* p, HINSTANCE h, UINT nServiceNameID, const GUID* plibid =
        NULL);
    void Start();
    void ServiceMain(DWORD dwArgc, LPTSTR* lpszArgv);
    void Handler(DWORD dwOpcode);
    void Run();
    BOOL IsInstalled();
    BOOL Install();
    BOOL Uninstall();
    LONG Unlock();
    void LogEvent(LPCTSTR pszFormat, ...);
    void SetServiceStatus(DWORD dwState);
    void SetupAsLocalServer();

// Implementation
private:
    static void WINAPI _ServiceMain(DWORD dwArgc, LPTSTR* lpszArgv);
    static void WINAPI _Handler(DWORD dwOpcode);

// data members
public:
    TCHAR m_szServiceName[256];
    SERVICE_STATUS_HANDLE m_hServiceStatus;
    SERVICE_STATUS m_status;
    DWORD dwThreadId;
    BOOL m_bService;
};

extern CServiceModule _Module;
#include <atlcom.h>
#include <comdef.h>
#include <atlwin.h>
```

```
////////////////////////////////////
```

```
// CRT
#include <assert.h>
#include <ctype.h>
#include <direct.h>

////////////////////////////////////
// STL
#include <locale>
#include <vector>
#include <map>
#include <sstream>
#include <fstream>
#include <list>
#include <string>
#include <sstream>
#include <algorithm>
using namespace std;
#define TOUPPER(str) ctype<string::value_type>().toupper(str.begin(), str.end())
#define TOLOWER(str) ctype<string::value_type>().tolower(str.begin(), str.end())

////////////////////////////////////
// SLMD
#include "LCEvMsgSrc.h"
#include "Logging.h"
extern CLogNTEvents      _logEvents;
extern CLogFile          _logFile;
extern CLogDebug         _logDebug;
extern CLogMulti         _logAll;

class CKioskServerApp;
extern CKioskServerApp  _theApp;

#define WMUSER_START      (WM_USER + 1000)

//({AFX_INSERT_LOCATION})
// Microsoft Visual C++ will insert additional declarations immediately before the
// previous line.

#endif // !defined(AFX_STDAFX_H__BF9A00C6-F03A-11D1-B49D-007060000000__INCLUDED_)
```

```
#include "stdafx.h"
#include "threadMain.h"
#include "filenameDelimited.h"
#include "threadReceiver.h"
#include "threadProcessor.h"

CKioskServerApp::CKioskServerApp()
{
    m_pthreadReceiver = NULL;
    m_pthreadProcessor = NULL;
}

BOOL CKioskServerApp::InitInstance()
{
    m_pthreadProcessor = new CThreadProcessor();
    if (!m_pthreadProcessor->CreateThread())
    {
        CLogMsgEvent msg(LCEV_GENERIC, SVRTY_ERROR);
        msg << "Unable to create processor thread in CKioskServerApp::InitInstance()";
        msg.Post(_logAll);
        return FALSE;
    }

    m_pthreadReceiver = new CThreadReceiver();
    if (!m_pthreadReceiver->CreateThread())
    {
        CLogMsgEvent msg(LCEV_GENERIC, SVRTY_ERROR);
        msg << "Unable to create receiver thread in CKioskServerApp::InitInstance()";
        msg.Post(_logAll);
        return FALSE;
    }

    while (!m_pthreadProcessor->PostThreadMessage(WMUSER_START, 0L, 0L))
        Sleep(100);

    while(!m_pthreadReceiver->PostThreadMessage(WMUSER_START, 0L, 0L))
        Sleep(100);

    return TRUE;
}

int CKioskServerApp::ExitInstance()
{
    if (m_pthreadReceiver != NULL)
        m_pthreadReceiver->PostThreadMessage(WM_QUIT, 0L, 0L);

    if (m_pthreadProcessor != NULL)
        m_pthreadProcessor->PostThreadMessage(WM_QUIT, 0L, 0L);

    return 0;
}

int CKioskServerApp::Run()
{
    return CWinThread::Run();
}
```

```
#ifndef _mainApp_h
#define _mainApp_h

class CKioskServerApp : public CWinApp
{
protected:
    CWinThread *      m_pthreadReceiver;
    CWinThread *      m_pthreadProcessor;

public:
    CKioskServerApp();
    virtual BOOL InitInstance();
    virtual int ExitInstance();
    virtual int Run();
};

#endif
```

```
// threadProcessor.cpp : implementation file
//

#include "stdafx.h"
#include "lckioskserver.h"
#include "threadProcessor.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// CThreadProcessor

IMPLEMENT_DYNCREATE(CThreadProcessor, CThreadServer)

CThreadProcessor::CThreadProcessor()
{
    useTimers(m_pnTimers, TIMER_MAX);
}

CThreadProcessor::~CThreadProcessor()
{
}

BOOL CThreadProcessor::InitInstance()
{
    // TODO: perform and per-thread initialization here
    return TRUE;
}

int CThreadProcessor::ExitInstance()
{
    // TODO: perform any per-thread cleanup here
    return CThreadServer::ExitInstance();
}

BEGIN_MESSAGE_MAP(CThreadProcessor, CThreadServer)
    //({AFX_MSG_MAP(CThreadProcessor)
    // NOTE - the ClassWizard will add and remove mapping macros here.
    //})AFX_MSG_MAP
    ON_THREAD_MESSAGE(WMUSER_START, onStart)
END_MESSAGE_MAP()

////////////////////////////////////
// CThreadProcessor message handlers

LRESULT CThreadProcessor::onStart(WPARAM wParam, LPARAM lParam)
{
    setTimer(TIMER_PROCESS, 10000);
    return FALSE;
}

void CThreadProcessor::onTimerIndex(int nIndex)
{
    killTimer(nIndex);

    switch (nIndex)
    {
    case TIMER_PROCESS:
        break;

    default:
        break;
    }
}
```

```
    return;  
}
```

```
#if !defined(AFX_THREADPROCESSOR_H__BF823571_E93B_11D3_B88C_CC792E000000__INCLUDED_)
#define AFX_THREADPROCESSOR_H__BF823571_E93B_11D3_B88C_CC792E000000__INCLUDED_
```

```
#include "threadServer.h"
```

```
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// threadProcessor.h : header file
//
```

```
////////////////////////////////////
// CThreadProcessor thread
```

```
class CThreadProcessor : public CThreadServer
```

```
{
    DECLARE_DYNCREATE(CThreadProcessor)
public:
    CThreadProcessor();
```

```
// Attributes
public:
```

```
// Operations
public:
```

```
// Overrides
    // ClassWizard generated virtual function overrides
    //{AFX_VIRTUAL(CThreadProcessor)
    public:
    virtual BOOL InitInstance();
    virtual int ExitInstance();
    //}AFX_VIRTUAL
```

```
// Implementation
```

```
protected:
    virtual ~CThreadProcessor();

    // Generated message map functions
    //{AFX_MSG(CThreadProcessor)
    // NOTE - the ClassWizard will add and remove member functions here.
    //}AFX_MSG
```

```
    DECLARE_MESSAGE_MAP()
```

```
protected:
    enum {TIMER_PROCESS, TIMER_MAX };

    UINT          m_pnTimers [TIMER_MAX];

    LRESULT onStart(WPARAM wParam, LPARAM lParam);

    virtual void onTimerIndex(int nIndex);
};
```

```
////////////////////////////////////
//{AFX_INSERT_LOCATION}
// Microsoft Visual C++ will insert additional declarations immediately before the
// previous line.

```

```
#endif // !defined(AFX_THREADPROCESSOR_H__BF823571_E93B_11D3_B88C_CC792E000000__INCLUDED_)
```

```

// threadReceiver.cpp : implementation file
//

#include "stdafx.h"
#include "lckioskserver.h"
#include "threadReceiver.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// CThreadReceiver

IMPLEMENT_DYNCREATE(CThreadReceiver, CThreadServer)

CThreadReceiver::CThreadReceiver()
{
    useTimers(m_pnTimers, TIMER_MAX);
}

CThreadReceiver::~CThreadReceiver()
{
}

BOOL CThreadReceiver::InitInstance()
{
    // TODO: perform and per-thread initialization here
    return TRUE;
}

int CThreadReceiver::ExitInstance()
{
    // TODO: perform any per-thread cleanup here
    return CThreadServer::ExitInstance();
}

BEGIN_MESSAGE_MAP(CThreadReceiver, CThreadServer)
    //{AFX_MSG_MAP(CThreadReceiver)
    // NOTE - the ClassWizard will add and remove mapping macros here.
    //}AFX_MSG_MAP
    ON_THREAD_MESSAGE(WMUSER_START, onStart)
END_MESSAGE_MAP()

////////////////////////////////////
// CThreadReceiver message handlers

LRESULT CThreadReceiver::onStart(WPARAM wParam, LPARAM lParam)
{
    setTimer(TIMER_RECEIVE, 30000);
    return FALSE;
}

void CThreadReceiver::onTimerIndex(int nIndex)
{
    killTimer(nIndex);

    switch (nIndex)
    {
    case TIMER_RECEIVE:
    default:
        break;
    }

    return;
}

```


}

```
#if !defined(AFX_THREADRECEIVER_H__BF823570_E93B_11D3_B88C_CC792E000000__INCLUDED_)
#define AFX_THREADRECEIVER_H__BF823570_E93B_11D3_B88C_CC792E000000__INCLUDED_
```

```
#include "threadServer.h"
```

```
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// threadReceiver.h : header file
//
```

```
////////////////////////////////////
// CThreadReceiver thread
```

```
class CThreadReceiver : public CThreadServer
{
    DECLARE_DYNCREATE(CThreadReceiver)
```

```
public:
    CThreadReceiver();
```

```
// Attributes
public:
```

```
// Operations
public:
```

```
// Overrides
    // ClassWizard generated virtual function overrides
    //{AFX_VIRTUAL(CThreadReceiver)
    public:
        virtual BOOL InitInstance();
        virtual int ExitInstance();
    //}AFX_VIRTUAL
```

```
// Implementation
```

```
protected:
    virtual ~CThreadReceiver();
```

```
    // Generated message map functions
    //{AFX_MSG(CThreadReceiver)
        // NOTE - the ClassWizard will add and remove member functions here.
    //}AFX_MSG
```

```
    DECLARE_MESSAGE_MAP()
```

```
protected:
    enum {TIMER_RECEIVE, TIMER_MAX };
    UINT      m_pnTimers [TIMER_MAX];

    LRESULT onStart(WPARAM wParam, LPARAM lParam);
    virtual void onTimerIndex(int nIndex);
};
```

```
////////////////////////////////////
```

```
{AFX_INSERT_LOCATION}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.
```

```
#endif // !defined(AFX_THREADRECEIVER_H__BF823570_E93B_11D3_B88C_CC792E000000__INCLUDED_)
```

```
#if !defined(AFX_THREADRECEIVER_H_BF823570_E93B_11D3_B88C_CC792E000000__INCLUDED_)
#define AFX_THREADRECEIVER_H_BF823570_E93B_11D3_B88C_CC792E000000__INCLUDED_
```

```
#include "threadServer.h"
```

```
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// threadReceiver.h : header file
//
```

```
////////////////////////////////////
// CThreadReceiver thread
```

```
class CThreadReceiver : public CThreadServer
{
    DECLARE_DYNCREATE(CThreadReceiver)
```

```
public:
    CThreadReceiver();
```

```
// Attributes
public:
```

```
// Operations
public:
```

```
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CThreadReceiver)
public:
    virtual BOOL InitInstance();
    virtual int ExitInstance();
//}}AFX_VIRTUAL
```

```
// Implementation
```

```
protected:
    virtual ~CThreadReceiver();

    // Generated message map functions
    //{AFX_MSG(CThreadReceiver)
    // NOTE - the ClassWizard will add and remove member functions here.
    //}{AFX_MSG
```

```
    DECLARE_MESSAGE_MAP()
```

```
protected:
    enum {TIMER_RECEIVE, TIMER_MAX };
    UINT      m_pnTimers [TIMER_MAX];

    LRESULT onStart(WPARAM wParam, LPARAM lParam);
    virtual void onTimerIndex(int nIndex);
};
```

```
////////////////////////////////////
```

```
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.
```

```
#endif // !defined(AFX_THREADRECEIVER_H_BF823570_E93B_11D3_B88C_CC792E000000__INCLUDED_
```

C:\Documents and Settings\billyhe\My ... \LCServices\LCKioskServer\threadServer.h 1

```
#if !defined(AFX_THREADSERVER_H__BF82356F_E93B_11D3_B88C_CC792E000000__INCLUDED_)
#define AFX_THREADSERVER_H__BF82356F_E93B_11D3_B88C_CC792E000000__INCLUDED_
```

```
#if _MSC_VER > 1000
#pragma Once
#endif // _MSC_VER > 1000
// threadServer.h : header file
//
```

```
////////////////////////////////////
// CThreadServer thread
```

```
class CThreadServer : public CWinThread
{
    DECLARE_DYNCREATE(CThreadServer)
protected:
    CThreadServer();          // protected constructor used by dynamic creation
```

```
// Attributes
public:
```

```
// Operations
public:
```

```
// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CThreadServer)
public:
    virtual BOOL InitInstance();
    virtual int ExitInstance();
//}}AFX_VIRTUAL
```

```
// Implementation
protected:
```

```
    virtual ~CThreadServer();

    // Generated message map functions
    {{{AFX_MSG(CThreadServer)
        // NOTE - the ClassWizard will add and remove member functions here.
    }}}AFX_MSG
```

```
    DECLARE_MESSAGE_MAP()
```

```
protected:
    static CCriticalSection    m_sync;

    UINT *                    m_pnTimers;
    int                        m_nMaxTimers;

    LRESULT onTimer(WPARAM wParam, LPARAM lParam);
    int getTimerIndex(UINT nTimerId);
    void killTimer(int nTimerIdx);
    void useTimers(UINT * pnTimers, int nMaxTimers);
    void setTimer(int nTimerIdx, unsigned long lMilliSecs);
```

```
    virtual void onTimerIndex(int nIdx);
```

```
public:
    static string            m_strWndClass;
};
```

```
////////////////////////////////////
```

```
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before this line.
```

previous line.

```
#endif // !defined(AFX_THREADSERVER_H_1E4441E4F6F6_4D7D81C0_92E00000_...INCLUDED_)
```

```

// threadServer.cpp : implementation file
//

#include "stdafx.h"
#include "lckioskserver.h"
#include "threadServer.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

string CThreadServer::m_strWndClass;
CCriticalSection CThreadServer::m_sync;

////////////////////////////////////
// CThreadServer

IMPLEMENT_DYNCREATE(CThreadServer, CWinThread)

CThreadServer::CThreadServer()
{
    m_pnTimers = NULL;
    m_nMaxTimers = 0;
}

CThreadServer::~CThreadServer()
{
}

BOOL CThreadServer::InitInstance()
{
    m_sync.Lock();
    if (m_strWndClass.size() == 0)
        m_strWndClass = AfxRegisterWndClass(0);
    m_sync.Unlock();

    return TRUE;
}

int CThreadServer::ExitInstance()
{
    // TODO: perform any per-thread cleanup here
    return CWinThread::ExitInstance();
}

BEGIN_MESSAGE_MAP(CThreadServer, CWinThread)
    //{{AFX_MSG_MAP(CThreadServer)
    // NOTE - the ClassWizard will add and remove mapping macros here.
    //}}AFX_MSG_MAP
    ON_THREAD_MESSAGE(WM_TIMER, onTimer)
END_MESSAGE_MAP()

LRESULT CThreadServer::onTimer(WPARAM wParam, LPARAM lParam)
{
    onTimerIndex(getTimerIndex(wParam));
    return FALSE;
}

void CThreadServer::onTimerIndex(int nIdx)
{
}

int CThreadServer::getTimerIndex(UINT nTimerId)
{
    for (int i = 0; i < m_nMaxTimers; i++)

```

```
{
    if (nTimerId == m_pnTimers[i])
        return i;
}

return -1;
}

void CThreadServer::killTimer(int nTimerIdx)
{
    ASSERT(nTimerIdx >= 0 && nTimerIdx < m_nMaxTimers);

    KillTimer(NULL, m_pnTimers[nTimerIdx]);
    m_pnTimers[nTimerIdx] = 0;
    return;
}

void CThreadServer::setTimer(int nTimerIdx, unsigned long lMilliSecs)
{
    ASSERT(nTimerIdx >= 0 && nTimerIdx < m_nMaxTimers);

    if (m_pnTimers[nTimerIdx])
        KillTimer(NULL, m_pnTimers[nTimerIdx]);

    m_pnTimers[nTimerIdx] = SetTimer(NULL, 0, lMilliSecs, NULL);
    return;
}

void CThreadServer::useTimers(UINT * pnTimers, int nMaxTimers)
{
    m_pnTimers = pnTimers;
    m_nMaxTimers = nMaxTimers;
    memset(m_pnTimers, 0, m_nMaxTimers * sizeof(UINT));
    return;
}
```